Pivotal

PRODUCT DOCUMENTATION

Spring Cloud Services for PCF Documentation

Version 1.4

Published: 5 Nov 2018

Rev: 01

Spring Cloud® Services

What Is Spring Cloud Services?

The rise in popularity of cloud-native architectures and the shift to implementing applications as series of focused microservices developed around bounded contexts has led to the discovery and rediscovery of patterns useful in designing distributed systems. Techniques such as service discovery, centralized configuration accessed through an application's deployment environment, and graceful degradation of behavior through circuit breakers are common solutions for this style of architecture and can be built into tools for applying these techniques across applications.

Netflix, a pioneer in the microservices space, has built many such tools. Eureka is a service registry, which registers all of a microservice's instances and supplies each microservice with instance information to use in discovering others. Hystrix, the Hystrix Dashboard, and Turbine provide fault tolerance using circuit breakers and enable monitoring of circuits across microservices and instances. These components have been battle-tested in a production environment facing some of the most demanding traffic requirements in the world, and are available as open-source software.

The Spring Cloud family of projects are based on Spring Boot and provide tools including Netflix's Eureka and Hystrix as libraries easily consumable by Spring applications, following idiomatic Spring conventions. Spring Cloud Netflix makes the use of Eureka and Hystrix as simple as including starters dependencies in an application and adding an annotation to a configuration class. Spring Cloud Config includes a configuration server and client library that enable Spring applications to consume centralized configuration as series of property sources.

Spring Cloud Services for Pivotal Cloud Foundry (PCF) packages server-side components of Spring Cloud projects, including Spring Cloud Netflix and Spring Cloud Config, and makes them available as services in the PCF Marketplace. This frees you from having to implement and maintain your own managed services in order to use the included projects. You can create a Config Server, Service Registry, or Circuit Breaker Dashboard service instance on-demand, bind to it and consume its functionality, and return to focusing on the value added by your own microservices.

Services

Spring Cloud Services currently provides the following services.

Service Type	Current Version
Config Server	8
Service Registry	8
Circuit Breaker Dashboard	8

See below for the Spring Cloud Services releases that include each service version.

Service Version	Tile Version
1	1.0.0 - 1.0.3
2	>= 1.0.4 & < 1.1.0
3	1.1.x
4	1.2.0 - 1.2.3
5	1.3.0
6	>= 1.2.4 & < 1.3.0
7	>= 1.3.1 & < 1.4.0
8	1.4.0 - 1.4.2
9	>= 1.4.3 & < 1.5.0

Dependencies

Spring Cloud Services relies on the following other Pivotal Cloud Foundry products.

Service	Supported Versions
MySQL	1.8.0-1.10.x
RabbitMQ	1.7.0-1.10.x

Product Snapshot

Current Spring Cloud Services for PCF Details

- **Version**: 1.4.8
- **Release Date**: 26 February 2018
- **Software component version**: Spring Cloud OSS Dalston.SR2
- **Compatible Ops Manager Version(s)**: 1.12.x, 1.11.x, 1.10.x, 1.9.x, 1.8.x
- **Compatible Elastic Runtime Version(s)**: 1.12.x, 1.11.x, 1.10.x, 1.9.x, 1.8.x
- **Supported IaaS:** All supported by PCF

Cloud Foundry CLI Plugin

Overview

The Spring Cloud Services plugin for the Cloud Foundry Command Line Interface tool (cf CLI) adds commands for interacting with Spring Cloud Services service instances. It provides easy access to functionality relating to the Config Server and Service Registry; for example, it can be used to send values to a Config Server service instance for encryption or to list all applications registered with a Service Registry service instance.

The Spring Cloud Services cf CLI plugin is open-source software released under the Apache 2.0 license. From the plugin's homepage on GitHub, you can raise issues or submit contributions for consideration by the plugin's authors.

Installation

To install the plugin, add the `CF-Community` repository to the cf CLI's list of plugin repositories:

```
$ cf add-plugin-repo CF-Community https://plugins.cloudfoundry.org


OK
https://plugins.cloudfoundry.org/list added as 'CF-Community'
```

Then install the `Spring Cloud Services` plugin (`SCSPlugin`) using the `cf install-plugin` command:

```
$ cf install-plugin -r CF-Community "Spring Cloud Services"
...
Plugin SCSPlugin v1.0.0 successfully installed.
```

Documentation

Documentation for the Spring Cloud Services cf CLI plugin's commands is available at the plugin's homepage on GitHub. See the Spring Cloud Services CF CLI Plugin Docs page.

Security Overview for Spring Cloud Services

Page last updated:

Glossary

The following abbreviations, example values, and terms are used in this topic as defined below.

- **CC**: The Cloud Foundry Cloud Controller.
- **CC DB**: The Cloud Controller database.
- **CF**: Cloud Foundry.
- **CF Application**: An application in Cloud Foundry; the `application` in `cf push application`.
- **Dashboard SSO**: Cloud Foundry's Dashboard Single Sign-On.
- **domain.com**: The value configured by an administrator for the Cloud Foundry system domain.
- **Eureka Application Name**: The identifier used by a CF Application to look up other CF Applications that are registered with a Spring Cloud Services Service Registry service instance. A CF Application that registers with a Spring Cloud Services Service Registry service instance will use the value of its `spring.application.name` property for this value by default.
- **FQDN**: Fully Qualified Domain Name.
- **Instance Record**: In Eureka, the core domain object; represents a CF Application that has registered with a Spring Cloud Services Service Registry service instance. An Instance Record is used to map a Virtual Hostname to a physical route (e.g. hostname, IP address, and port). An Instance Record can have metadata associated with it.
- **Operator**: A user of Pivotal Cloud Foundry® Operations Manager.
- **SB**: The Spring Cloud Services Service Broker.
- **SCS**: Spring Cloud Services.
- **UAA**: The Cloud Foundry User Account and Authentication Server.

A Note on TLS/SSL

All components in SCS force HTTPS for all inbound requests. Because SSL terminates at the load balancer, requests are deemed to originate over HTTPS by inspecting the `X-Forwarded-For`, `X-Forwarded-Proto`, and `X-Forwarded-Port` headers. More on how this is implemented can be found in the Spring Boot documentation.

All outbound HTTP requests made by SCS components are also made over HTTPS. These requests nearly always contain credentials, and are made to other components within the Cloud Foundry environment. In an environment that uses self-signed certificates in the HA proxy configuration, outbound HTTPS requests to other components would fail because the self-signed certificate would not be trusted by the JVM's default truststore (e.g. `cacerts`). To get around this, all components use the cloudfoundry-certificate-truster to add the CF environment's SSL certificate to the JVM's truststore on application startup.

The SSL certificate used by the CF environment must also include Subject Alternative Names (SANs) for domain wildcards `*.login.domain.com` and `*.uaa.domain.com`. This is needed for the correct operation of Multi-tenant UAA, which relies on subdomains of UAA.

A Note on Multi-Tenant UAA

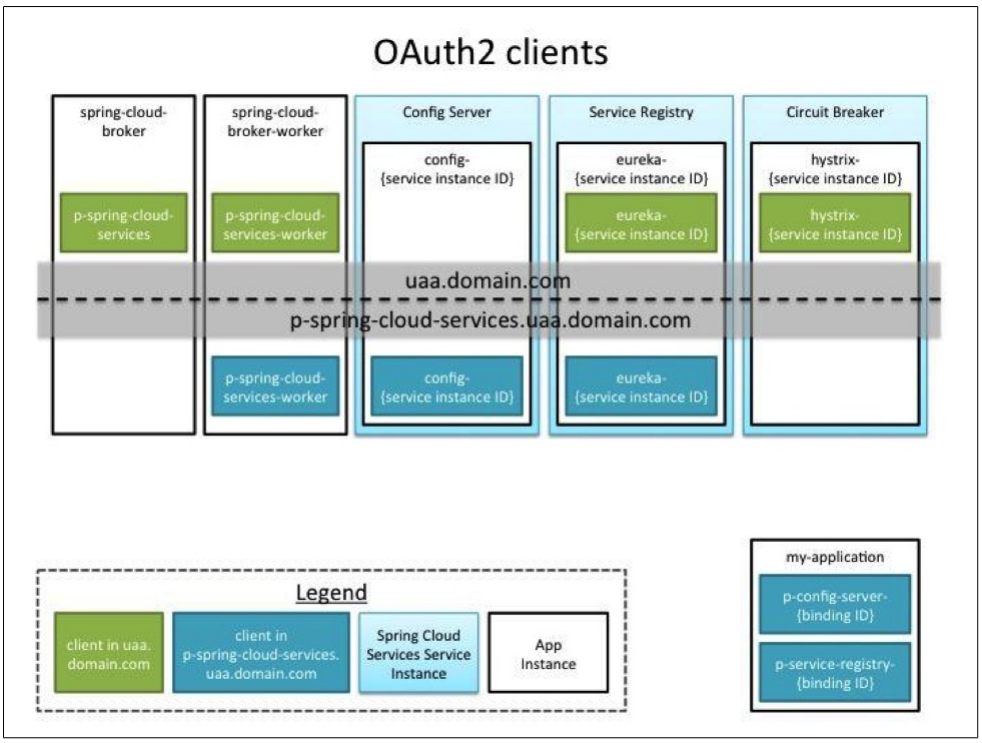
SCS uses the multi-tenancy features of UAA to implement "the principle of least privilege". This is required because SCS creates and deletes OAuth clients when applications bind and unbind to service instances. These clients must also have non-empty authorities. In order to create clients with arbitrary authorities, the actor must have the scopes `uaa.admin` and `clients.write`; essentially they must be the admin.

Making admin-level credentials (e.g. the UAA admin client) accessible to an application is dangerous and should be avoided. For example, if one were to break into that application and get those credentials, the credentials could be used to affect the entire CF environment. This is why SCS operates in two UAA domains (also called Identity Zones).

The platform Identity Zone (e.g. `uaa.domain.com`) contains the users of SCS (e.g. Space Developers) and the Dashboard SSO clients used by each Service Instance. An admin-level client is not required for creating and deleting SSO clients because the SSO client's scope values are fixed and only the most basic `uaa.resource` authority is needed by the SSO client.

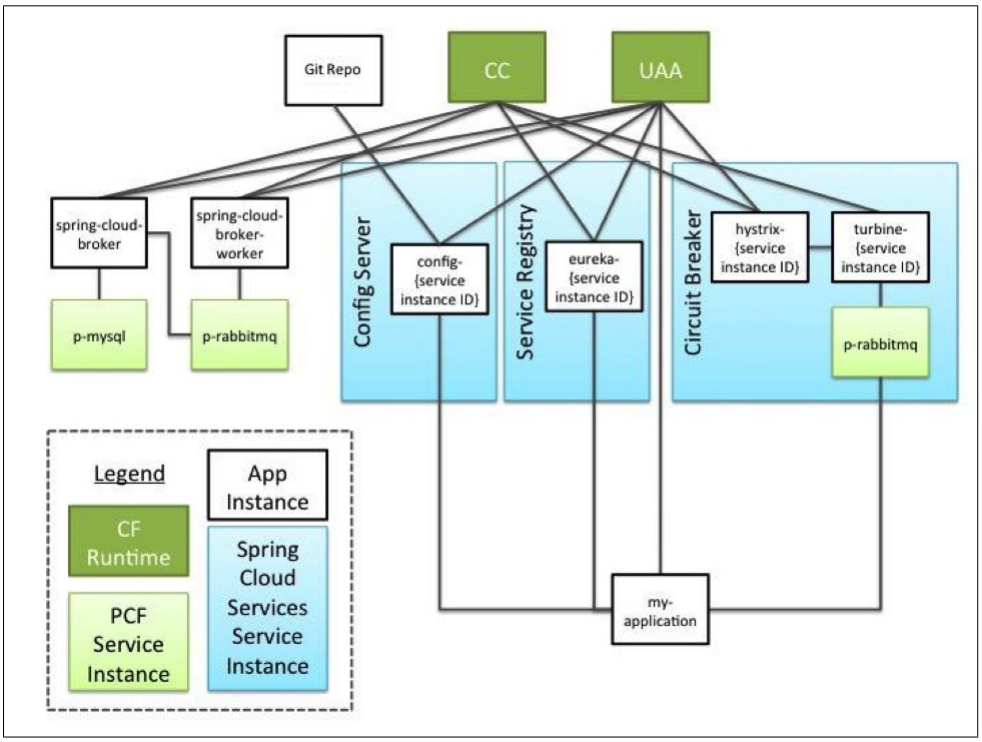
The other Identity Zone is specific to SCS (e.g. `p-spring-cloud-services.uaa.domain.com`). This Identity Zone contains no users, but it contains the clients used by bound applications to access protected resources on SCS service instances. These clients require dynamic authority values, which can only be created by an admin-level actor. This admin-level actor (the p-spring-cloud-services-worker client) also exists in the Spring Cloud Services Identity Zone, but because it exists in this Identity Zone, it cannot affect anything in the platform Identity Zone. If these credentials were leaked, the damage would be limited to SCS.

The following diagram illustrates the OAuth clients used by SCS and the Identity Zones in which they exist.



Spring Cloud Services Overview

Before continuing, you should familiarize yourself with the following diagram, which shows the direct lines of communication between application components.



Core Components

The Installation File

The installation file is the `.pivotal` file that contains the Spring Cloud Services product.

The installation file is generated from source code repositories hosted on `github.com`. The repositories are can only be accessed by authorized persons and cannot be accessed over a non-encrypted channel. No credentials are stored in any source repository used by Spring Cloud Services.

Concourse is used as the build pipeline to generate the installation file. The infrastructure that hosts Concourse can only be accessed from within the Pivotal network, using a username and password stored internally by Concourse. The build pipeline configuration contains credentials to the S3 bucket which hosts the installation file and intermediate artifacts, as well as the GitHub API key providing read/write access to the source code repositories. The resulting installation file built by the Concourse pipeline does not contain any credentials.

The installation file is downloaded from https://network.pivotal.io, can only be accessed by authorized persons, and cannot be accessed over a non-encrypted channel.

To install Spring Cloud Services, the installation file must be uploaded to Pivotal Cloud Foundry® Operations Manager. Ops Manager can only be accessed over HTTPS by persons knowing the username and password for Ops Manager.

Spring Cloud Service Broker

The Spring Cloud Service Broker is responsible for receiving Service Broker API requests from the Cloud Controller (CC), and hosts the primary dashboard UI for all service types.

Deployment Details

Org: `system`
Space: `p-spring-cloud-services`
CF Application Name: `spring-cloud-broker`

Entry Points

The Service Broker API

Service Broker API endpoints require HTTPS and Basic authentication. Valid credentials for Basic authentication are made available to the SB via environment variables, and are therefore stored in the CC database in encrypted form. The CC also makes Service Broker API requests, and the credentials used to make these API requests are stored in the CC DB in encrypted form as well.

Service Broker Dashboard

Dashboard endpoints require HTTPS and an authenticated session. Unauthenticated requests are redirected to UAA to initiate the OAuth 2 Authorization Code flow as documented here; for the flow to complete, the user must be authenticated with UAA. Once the token is obtained, the session is authenticated. The token is then used to access the CC API to check permissions on the service instance being accessed, as documented here. This flow is typical of Dashboard Single Sign-On for Service Instances.

The service instance ID used in the permission check is the first GUID that is present in the URL, and this value is also used by the rest of the SB application to identify the service instance that the user is trying to access.

GET /dashboard/instance/{serviceInstanceId}

This endpoint returns service instance details such as service name, plan ID, org, and space, as well as credentials that need to be propagated to applications bound to this service instance. Because the service instance permission check is performed, this endpoint is only accessible to Space Developers who can bind applications to this service instance. Once an application is bound to this service instance, a Space Developer can see the same credentials returned by this endpoint in the bound application's binding credentials (e.g. via Pivotal Cloud Foundry Applications Manager.)

GET|POST /dashboard/instance/{serviceInstanceId}/env

This endpoint gets or modifies environment variables for the service instance's backing application. Environment variables that can be viewed or modified with this endpoint must be whitelisted; this whitelisting can only be modified by the Operator or the developers of Spring Cloud Services. Each service type has its own whitelist, and the only service type with whitelisted environment variables is Config Server. The Config Server's whitelisted environment variables specify the repository type, the URI of the repository, and the username and password used to access the repository. Any POST to this endpoint will result in a restart of the backing application, regardless of whether any environment variables were actually modified.

GET /dashboard/instance/{serviceInstanceId}/health

This endpoint returns the response of the `/health` endpoint on the backing application. The `/health` endpoint is provided by Spring Boot Actuator and may be accessed anonymously for summary information or using OAuth2 authentication for detailed information. This is used by the Config Server configuration page to indicate if there are any problems with the Config Server's configuration.

Caching the Service Instance Permission Check

The service instance permission check is cached in session (if available). The cache is used only during GET, HEAD, and OPTIONS requests, with the exception of the `/instances/{service_instance_guid}/env` endpoint. Any request for this endpoint does not use the cache, because this endpoint exposes application environment variables—and, in the case of Config Server, the username and password used to access the Config Server's backing repository.

Caching the permission check in this manner avoids the load and extra round trip to the CC for doing the permission check, while still checking permissions before performing any critical operation.

Actuator Endpoints

Actuator endpoints are enabled and require HTTPS and an authenticated session. The token is obtained from UAA via the OAuth 2 Authorization Code flow as documented here; the token is then used to make a request to the CC `/v2/apps/:guid/env` endpoint, which can only be accessed by Space Developers in the space that hosts the SB. The GUID used in the request is extracted from the `VCAP_APPLICATION` environment variable, whose value is given by the CC. The only user that can access these endpoints is the Operator.

This Actuator SSO flow is used by other components that already use Dashboard SSO, as no additional dependencies are required to enable this.

External Dependencies

p-mysql

The SB uses a p-mysql service instance to store information about SCS service instances. The most sensitive information stored here is credentials for the p-rabbitmq instances used for the Circuit Breaker service instances, which could be used to disable the Hystrix dashboard as described in app turbine-{guid} > External Dependencies > p-rabbitmq. The only way to access the data in this p-mysql service instance is through the SB's binding credentials or directly from the filesystem, and these would only be accessible to the Operator.

p-rabbitmq

The SB uses a p-rabbitmq service instance to communicate with the spring-cloud-broker-worker application (the Worker). The messages passed propagate the provision, deprovision, bind, and unbind requests received by the SB API to the Worker, as well as getting and setting application environment variables for service instance backing applications in the "instances" space. The credentials for accessing the p-rabbitmq service instance are provided through the SB's `VCAP_SERVICES` environment variable as a result of binding to the service instance. The credentials would only be visible to a Space Developer in the "p-spring-cloud-services" space, and only the Operator would have this access.

The Cloud Controller

The SB communicates to the CC via HTTPS to perform permission checks using the current user's token as described in the Service Broker Dashboard section, and to derive URIs for UAA. The URI to the CC is set at installation time in the SB's environment variables, under the name `CF_TARGET`. The environment variable could be modified so that the user's token is sent to another party. The environment variable can only be modified by a Space Developer in the "p-spring-cloud-services" space, and only the Operator would have this access.

UAA

The SB communicates with UAA via HTTPS to enable Dashboard SSO. All requests to UAA are authenticated as per the OAuth 2 specification, using the p-spring-cloud-services client credentials. The URI to UAA is derived from the CC API `/v2/info` endpoint, whose threat mitigations are described in the previous section.

OAuth Clients

p-spring-cloud-services

> **Identity Zone**: UAA
> **Grant Type**: Authorization Code
> **Redirect URI**: https://spring-cloud-broker.domain.com/dashboard/login
> **Scopes**: `cloud_controller.read`, `cloud_controller.write`, `cloud_controller_service_permissions.read`, `openid`
> **Authorities**: `uaa.resource`
> **Token expiry**: default (12 hours)

This client is used for SSO with UAA. The scopes allow the SB to access the Cloud Controller API on the user's behalf, and are auto-approved. The `uaa.resource` authority allows the SB to access the token verification endpoints on UAA. The client ID and client secret are made available to the SB via environment variables, which are set at installation time and are stored in encrypted form in the CC DB.

Spring Cloud Service Broker Worker

The Spring Cloud Service Broker Worker (the Worker) is responsible for managing service instance backing applications. Some of this work is done asynchronously from the requests that initiate it.

Deployment Details

> **Org**: `system`
> **Space**: `p-spring-cloud-services`
> **CF Application Name**: `spring-cloud-broker-worker`

Entry Points

The RabbitMQ Message Listener

The Worker listens for messages posted to the p-rabbitmq service instance. The SB posts these messages as described in Spring Cloud Service Broker > External Dependencies > p-rabbitmq. Only the Operator, the SB, and the Worker can access this instance of p-rabbitmq.

Actuator Endpoints

Actuator endpoints are enabled and require HTTPS and OAuth 2 authentication.

External Dependencies

p-rabbitmq

The Worker uses a p-rabbitmq service instance to receive messages from the SB. The messages received originate from the provision, deprovision, bind, and unbind requests received by the SB API, as well as from the `/dashboard/instance/{serviceInstanceId}/env` endpoint. The credentials for accessing the p-rabbitmq service instance are provided through the Worker's `VCAP_SERVICES` environment variable as a result of binding to the service instance. The credentials would only be visible to a Space Developer in the "p-spring-cloud-services" space, and only the Operator would have this access.

The Cloud Controller

The Worker communicates to the CC via HTTPS to carry out management tasks for service instance backing apps. All backing applications reside in the "p-

spring-cloud-services" org, "instances" space. The Worker uses its own user credentials to obtain a token for accessing the CC API and acts as a Space Developer in the "instances" space.

The URI to the CC is set at installation time in the Worker's environment variables, under the name `CF_TARGET`, which is set at installation time and can only be modified by the Operator.

UAA

The Worker uses the client management APIs of UAA to create and delete clients in both the platform and Spring Cloud Services Identity Zones. All requests are authenticated by including the OAuth 2 bearer token in the request, and are made over HTTPS. The credentials used in these requests are detailed in the OAuth Clients section. The URI to UAA is derived from the CC API `/v2/info` endpoint.

OAuth Clients

p-spring-cloud-services-worker

Identity Zone: UAA
Grant Type: Client Credentials
Redirect URI: https://spring-cloud-broker.domain.com/dashboard/login
Scopes: `cloud_controller.read`, `cloud_controller_service_permissions.read`, `openid`
Authorities: `clients.write`
Token expiry: default (12 hours)

This client is used to create and delete SSO clients for each Service Registry and Circuit Breaker Dashboard service instance, which use Dashboard SSO for their own dashboards. The grant type is Client Credentials, which allows the Worker to act on its own. The `clients.write` authority allows the Worker to create and delete clients, but this client is limited in the kinds of clients which it can create. The scopes that are registered allow this client to create other clients with the same scopes, but these clients cannot have any authorities. In other words, clients created via this client cannot act on their own, and can only act on behalf of a user within the scopes that are registered for this client. This limitation prevents privilege escalation.

While arbitrary clients cannot be created via this client, this client is able to delete any client in the UAA Identity Zone due to the `clients.write` authority.

The client ID and client secret are made available to the SB via environment variables, which are set at installation time and are stored in encrypted form in the CC DB. The credentials are only visible to the Operator.

p-spring-cloud-services-worker

Identity Zone: Spring Cloud Services
Grant Type: Client Credentials
Authorities: `clients.read`, `clients.write`, `uaa.admin`
Token expiry: default (12 hours)

This client is used to create and delete clients in the Spring Cloud Services Identity Zone for each Config Server and Service Registry service instance. The grant type is Client Credentials, which allows the Worker to act on its own. The `uaa.admin` authority allows the Worker to create and delete clients in the Spring Cloud Services Identity Zone without any limitations, unlike the corresponding client in the UAA Identity Zone. However, these privileges do not extend into the UAA Identity Zone.

The client ID and client secret are the same as the corresponding client in the UAA Identity Zone.

Users

p-spring-cloud-services

This user is a Space Developer in the "p-spring-cloud-services" org, "instances" space. The user has no other roles. The username and password are made available to the Worker via environment variables, which are set at installation time and are stored in encrypted form in the CC DB. The credentials are only visible to the Operator.

Config Server Service Instances

Config Server

The Config Server CF Application is a backing application for a Config Server service instance. This application is responsible for handling requests for configuration values from bound applications.

Deployment Details

Org: `p-spring-cloud-services`
Space: `instances`
CF Application Name: `config-[guid]`, where `[guid]` is the service instance ID

Entry Points

Config Server REST endpoints

The following entry points are defined by Spring Cloud Config Server.

GET /{application}/{profile}[/{label}]

This returns configuration values for a requesting application. The request must be authenticated with an OAuth 2 Bearer token issued by the Spring Cloud Services Identity Zone. The following claims are checked:

- `aud`: must equal `config-server-[guid]`
- `iss`: must equal `https://p-spring-cloud-services.uaa.domain.com/oauth/token`
- `scope`: must equal `config-server-[guid].read`

The GUID in the `aud` and `scope` values must match the `SERVICE_INSTANCE_ID` environment variable that is set in the Config Server application at provision time.

GET /health

This is used by the Dashboard UI for checking if the Config Server is configured properly. Requests sent to this endpoint using an OAuth 2 Bearer token receive a JSON object response containing health details for each configuration source. The following example shows a typical response from an authenticated request to a health endpoint; in this example, the Config Server is using a Git confoguration source:


```
{
 "status": "UP",
 "git": {
  "status": "UP",
  "default": {
   "status": "UP",
   "repository": {
    "uri": "https://github.com/spring-cloud-services-samples/cook-config"
   }
  },
  "cook": {
   "status": "UP",
   "repository": {
    "uri": "https://github.com/spring-cloud-services-samples/cook-config"
   }
  }
 },
 "diskSpace": {
  "status": "UP",
  "total": 1056858112,
  "free": 889167872,
  "threshold": 10485760
 },
 "refreshScope": {
  "status": "UP"
 }
}
```


The overall health status can only be `UP` if all of the configuration source health indicators report `UP`. In the example response below, the Config Server is using a composite of configuration sources, and the Vault source's health status is `DOWN`. This in turn leads to the overall summary health status for the Config Server being set to `DOWN`:


```
{
 "status": "DOWN",
 "git": {
  "status": "UP",
  "git-1": {
   "status": "UP",
   "repository": {
    "uri": "https://github.com/spring-cloud-services-samples/cook-config"
   }
  },
  "cook": {
   "status": "UP",
   "repository": {
    "uri": "https://github.com/spring-cloud-services-samples/cook-config"
   }
  }
 },
 "vault": {
  "status": "DOWN",
  "vault-1": {
   "status": "DOWN",
   "server": {
    "scheme": "http",
    "port": 8200,
    "host": "myvault.mycompany.com"
   },
   "error": "Unable to contact Vault server at http://myvault.mycompany.com:8200/v1/sys/health",
   "reason": "I/O error on GET request for \"http://myvault.mycompany.com:8200/v1/sys/health\": null; nested exception is org.apache.http.client.ClientProtocolException"
  }
 },
 "diskSpace": {
  "status": "UP",
  "total": 1056858112,
  "free": 889167872,
  "threshold": 10485760
 },
 "refreshScope": {
  "status": "UP"
 }
}
```


The endpoint may also be accessed anonymously, in which case it will only return `{"status":"UP"}` or `{"status":"DOWN"}`.

OAuth 2 Clients for Bound Applications

To provide access to bound applications, each SB bind request creates the following OAuth 2 client:

Client ID: p-config-server-[bindingId]
Identity Zone: Spring Cloud Services
Grant Type: Client Credentials

Scopes: `uaa.none`
Authorities: `p-config-server-[guid].read`
Token expiry: 30 seconds

The client ID and client secret for this client are provided to the bound application via the binding credentials in the `VCAP_SERVICES` environment variable. Only a Space Developer in the space that contains the bound application would have access to this.

No Application-Level Access Control

Other than checking for the claims mentioned in the previous section, no other attributes are considered when making an access control decision. Because of this, any application bound to the service instance will be able to access any configuration served by this Config Server.

Actuator Endpoints

Actuator endpoints are enabled and require OAuth 2 authentication, with the exception of `/health` as described above; the `health` endpoint can also be accessed anonymously to receive less-detailed information.

External Dependencies

Git repository

The actual configurations are stored in a Git repository and retrieval of these configurations by the Config Server is subject to the Git repository's security requirements. The URI and the username and password (if applicable) are set by the Space Developer that configures the service instance. The UI gets and sets these credentials through the Service Broker's `/dashboard/instances/{guid}/env` endpoint, so that the credentials are made available to the Config Server via environment variables (and hence stored encrypted in the CC DB).

UAA

The Config Server accesses the `/token_key` endpoint on UAA to retrieve the token verification key. This key material is used to verify the JWT signature of the token in order to authenticate the request.

OAuth Clients

config-server-{guid}

Identity Zone: Spring Cloud Services
Grant Type: Client Credentials
Scopes: `uaa.none`
Authorities: `uaa.resource`
Token expiry: default (12 hours)

The `uaa.resource` authority allows the config-server application to access the token verification endpoints on UAA. The client is created by the p-spring-cloud-services-worker client at provision time. The client ID and client secret are made available to the config-server application via environment variables set at provision time, and are only visible to the Operator.

Service Registry Service Instances

Eureka

The Eureka CF Application is a backing application for Service Registry service instances. This application hosts a REST API for the Service Registry and a dashboard UI for viewing the status of the Service Registry.

Deployment Details

Org: `p-spring-cloud-services`
Space: `instances`
CF Application Name: `eureka-[guid](-node[n])`, where `[guid]` is the service instance ID. When running in HA mode, `(-node[n])` uniquely identifies the application node.

Roles

Clients are categorized into three roles, and the authorization model is designed around these three roles.

Node

When a Service Registry is deployed in High Availability mode, the Node role represents other Eureka servers that are part of the same service instance as this server.

For an incoming request to be identified as coming from a Node, the request must contain a token with the following claims:

- `iss`: must be the same issuer as this server
- `sub`: must be the same `client_id` as this server
- `scope`: must equal `p-service-registry-[guid].read`, `p-service-registry-[guid].write`, where `[guid]` is the service instance ID of this server

Peer

When a Service Registry is configured for peer replication with other service instances, the Peer role represents one of the Eureka servers of a configured peer service instance.

For an incoming request to be identified as coming from a Peer, the request must contain a token with the following claims:

- `iss`: must be the issuer as reported by the peer's `/info` endpoint
- `sub`: must be the hostname portion of the configured peer's FQDN
- `scope`: must equal `p-service-registry-[peer-guid].read`, `p-service-registry-[peer-guid].write`, where `[peer-guid]` is the GUID contained in the peer's hostname

Bound Application

A Bound Application is a CF Application bound to this service instance. It does not represent a CF Application bound to a Peer service instance, as a Bound Application is only authorized to communicate with the service instance to which it is bound.

For an incoming request to be identified as coming from a Bound Application, the request must contain a token with the following claims:

- `iss`: must be the same issuer as this server
- `sub`: must start with `p-service-registry`
- `scope`: must equal `p-service-registry-[guid].read`, `p-service-registry-[guid].write`, where `[guid]` is the service instance ID of this server

Entry Points

Eureka REST endpoints

GET /info

This endpoint returns basic information about the server's configuration as part of the handshake process before a Peer can replicate state with the service instance. Authentication is not required to access this endpoint. The endpoint returns the URIs of the service instance's configured peers, the URI of the service instance's token issuer, the service instance's node count, and the service instance's version.

POST /eureka/peerreplication/batch/

This endpoint receives batches of replicated requests from other Eureka servers. Only requests from Nodes and Peers are accepted.

*GET|POST|PUT|DELETE /eureka/** (all other endpoints)*

These remainder of the REST endpoints can be found in the Netflix Eureka documentation. These endpoints can only be accessed by Nodes or Bound Applications.

Record Level Access Control

In addition to checking token claims, further checks are done on POST, PUT, and DELETE requests to ensure that a Eureka Instance Record can only be modified or deleted by the CF Application that created it. In addition, Spring Cloud Services enforces restrictions on the registration of Eureka Application Names within a Eureka cluster (i.e. a service instance and its configured peers, if any).

Instance Record Ownership

The Eureka server enforces ownership of Instance Records to prevent binding credentials from being misused to affect other Instance Records, especially those originating from CF Applications in other spaces. To ensure that a Instance Record can only be modified by the CF Application that created it, the Eureka server has been modified to record the CF Application's identity, which is conveyed by the `sub` claim of the Bound Application's token. This value is saved in the Instance Record's metadata under the key `registrant_principal` upon registration. Any request to modify an Instance Record must contain a token whose `sub` claim matches the `registrant_principal`. Also, the `registrant_principal` metadata field cannot be modified after the Instance Record is created.

Application Name Ownership

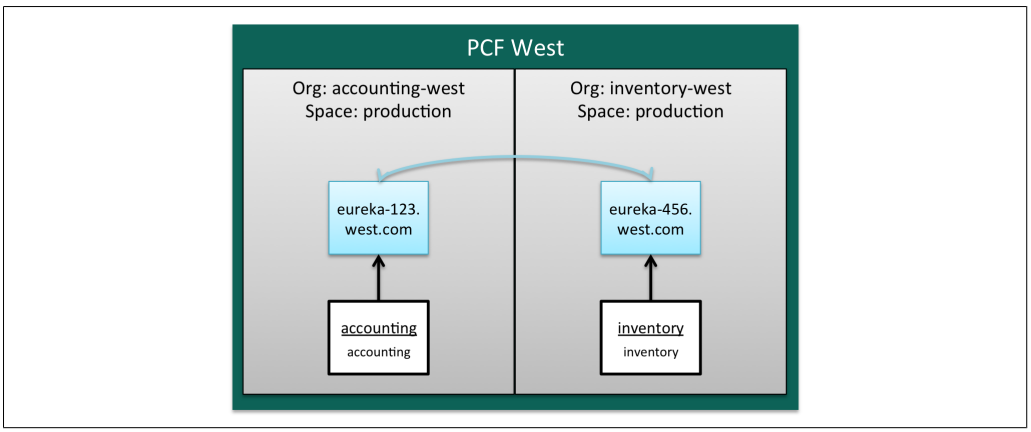
The Eureka server enforces the ownership of Eureka Application Names within a cluster. For a given Eureka cluster, possibly comprised of peers spanning multiple PCF installations, only one service instance in a PCF installation may register instances with a particular Eureka Application Name. This is enforced to ensure that CF Applications in other spaces cannot register and draw traffic away from a given application, as a form of man-in-the-middle attack. However, a service instance in a different PCF installation can still register the same Eureka Application Name. This is allowed so that CF Applications in other PCF installations can be used for failover, in case all instances with a given Eureka Application Name in a particular PCF are down.

To enforce Eureka Application Name ownership, the URI of the Eureka server that registers the Instance Record is recorded under the metadata key `registrar_uri`. A portion of this URI is used to identity the PCF installation that hosts the Eureka server. When a CF Application registers with the server by submitting an Instance Record, all other Instance Records with the same Eureka Application Name are inspected. If an Instance Record already exists in the same PCF installation as the server receiving the registration request, and this Instance Record's `registrar_uri` does not match the server's, the registration request is rejected.

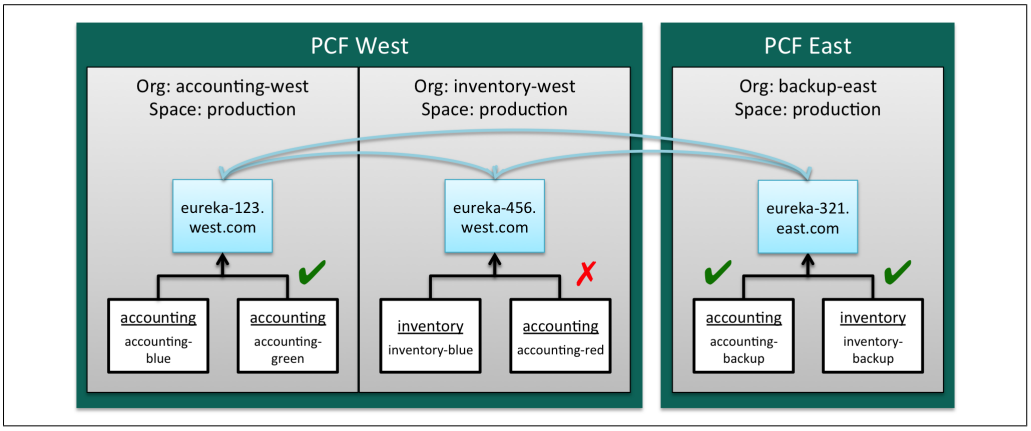
The ownership of Eureka Application Names by a service instance parallels the ownership of routes by a space in CF. However, Eureka Application Names cannot be reserved by a service instance at this time, in contrast to the creation of a route in a space, where the route is not yet bound to a CF Application. If one service instance were to register Instance Records with a particular Eureka Application Name and then all such Instance Records became de-registered (through either planned or unplanned outages), another CF Application in a different space could register with that same Eureka Application Name by binding with a peer service instance. This would prevent the original CF Application(s) from registering again when they came back online.

Eureka Application Name Ownership Example

PCF West is a PCF installation with Service Registry service instances in different organizations. These service instances are configured for peer replication with one another. One space hosts a CF Application `accounting` and the other hosts a CF Application `inventory`. They both register with Eureka Application Names `accounting` and `inventory` as underlined, respectively.



Later on, the `accounting-west/production` space wants to deploy `accounting` using a Blue/Green deployment strategy. The `accounting` CF Application is renamed `accounting-blue`, but its Eureka Application Name stays the same. Then `accounting-green` is pushed, bound to the Service Registry service instance, and started. On startup it successully registers with the Eureka Application Name `accounting`. This is allowed because there are no other service instances in this cluster, in PCF West, that have registered the Eureka Application Name `accounting`.



A space developer with access to the `inventory-west/production` space wishes to deploy a CF Application `accounting-red` in that space and register it with the Eureka Application Name `accounting`. When `accounting-red` starts up, it tries to register but its registration requests are rejected. This is because the Eureka Application Name `accounting` has already been registered with another service instance in the cluster, in the same PCF installation.

PCF East comes online, and provides failover for `accounting` and `inventory` apps. This is allowed because there are no other service instances in this cluster, in PCF East, that have registered the Eureka Application Names `accounting` or `inventory`.

OAuth 2 Clients for Bound Applications

To provide access to the Eureka REST API for bound applications, each SB bind request creates the following OAuth 2 client:

Client ID: p-service-registry-[bindingId]
Identity Zone: Spring Cloud Services
Grant Type: Client Credentials
Scopes: `uaa.none`
Authorities: `p-service-registry-[guid].read`, `p-service-registry-[guid].write`
Token expiry: 30 seconds

The client ID and client secret for this client is provided to the bound application via the binding credentials in the `VCAP_SERVICES` environment variable. Only a Space Developer in the space that contains the bound application would have access to this.

Eureka Dashboard

The Eureka Dashboard requires the typical Dashboard SSO flow. The service instance ID used in the permission check is supplied by an environment variable set at provision time by the Worker. The service instance permission check is cached in the same manner as the checks done by the service broker.

Actuator Endpoints

Actuator endpoints require the same Actuator SSO flow used by the Service Broker.

External Dependencies

UAA

The Eureka Server accesses the `/token_key` endpoint on UAA to retrieve the token verification key. This key material is used to verify the JWT signature of the token in order to authenticate the request.

The Cloud Controller

The SB communicates to the CC via HTTPS to perform permission checks using the current user's token, as described in the Entry Points section, and to derive URIs for UAA. The URI to the CC is set at installation time in the SB's environment variables, under the name `CF_TARGET`. The environment variable could be modified so that the user's token is sent to another party. The environment variable can only be modified by a Space Developer in the "p-spring-cloud-services" space, and only the Operator would have this access.

OAuth Clients

eureka-{guid}

Identity Zone: UAA
Grant Type: Authorization Code
Redirect URI: https://eureka-{guid}.domain.com/dashboard/login
Scopes: `openid`, `cloud_controller.read`, `cloud_controller_service_permissions.read`
Authorities: `uaa.resource`
Token expiry: default (12 hours)

This client is used for Dashboard SSO with UAA. The scopes allow the Eureka application to access the Cloud Controller API on the user's behalf, and are auto-approved. The `uaa.resource` authority allows the Eureka application to access the token verification endpoints on UAA. The client is created by the p-spring-cloud-services-worker client at provision time. The client ID and client secret are made available to the Eureka application via environment variables set at provision time.

eureka-{guid}

Identity Zone: Spring Cloud Services
Grant Type: Client Credentials
Scopes: `uaa.none`
Authorities: `p-service-registry-[guid].read`, `p-service-registry-[guid].write`, `uaa.resource`
Token expiry: default (12 hours)

The `uaa.resource` authority allows the Eureka application to access the token verification endpoints on UAA. The other authorities allow it to access other members of the cluster. The client is created by the p-spring-cloud-services-worker client at provision time. The client ID and client secret are made available to the Eureka application via environment variables set at provision time. The credentials are the same as the corresponding client in the UAA Identity Zone.

Circuit Breaker Dashboard Service Instances

Hystrix

The Hystrix application is one of two backing applications for a Circuit Breaker Dashboard service instance. This application is derived from the Netflix Hystrix Dashboard.

Deployment Details

Org: `p-spring-cloud-services`
Space: `instances`
CF Application Name: `hystrix-[guid]`, where `[guid]` is the service instance ID

Entry Points

Hystrix Dashboard

The Hystrix Dashboard requires the typical Dashboard SSO flow. The service instance ID used in the permission check is supplied by an environment variable set at provision time by the Worker. The service instance permission check is cached in the same manner as the checks done by the service broker.

Actuator Endpoints

Actuator endpoints require the same Actuator SSO flow used by the Service Broker.

External Dependencies

Turbine

The Hystrix Dashboard UI uses an EventSource in the browser to listen to Server Sent Events emitted by the Turbine application. Because the Turbine application's origin differs from the Hystrix application, and EventSources do not support CORS, the Hystrix application must handle the EventSource request and proxy this request to the Turbine application.

The Turbine application must authenticate this request, so the Hystrix application includes the token in the proxy request. The proxy request is handled on Hystrix via the endpoint `/proxy.stream?origin=[turbine url]`. This endpoint on open-source Hystrix can be used as an open proxy, so to protect the token from being leaked, the Turbine URL in the origin query parameter must match the `TURBINE_URL` environment variable of the Hystrix application. This variable is set at provision time, always begins with `https://`, and can only be modified by the Operator.

UAA

The Hystrix application accesses the `/token_key` endpoint on UAA to retrieve the token verification key. This key material is used to verify the JWT signature of the token in order to authenticate the request.

The Cloud Controller

The Hystrix application communicates to the CC via HTTPS to perform permission checks using the current user's token, as described in the Entry Points section, and to derive URIs for UAA. The URI to the CC is set by the Worker at provision time in the application's environment variables, under the name `CF_TARGET`. The environment variable can only be modified by the Operator.

OAuth Clients

hystrix-{guid}

> **Identity Zone**: UAA
> **Grant Type**: Authorization Code
> **Scopes**: `openid`, `cloud_controller.read`, `cloud_controller_service_permissions.read`
> **Authorities**: `uaa.resource`
> **Token expiry**: default (12 hours)

This client is used for SSO with UAA. The scopes allow the Hystrix application to access the Cloud Controller API on the user's behalf, and are auto-approved. The `uaa.resource` authority allows the Hystrix application to access the token verification endpoints on UAA. The client is created by the p-spring-cloud-services-worker client at provision time. The client ID and client secret are made available to the Hystrix application via environment variables set at provision time.

Turbine

The Turbine application is the other backing application for a Circuit Breaker Dashboard service instance. This application is derived from Netflix Turbine, which aggregates Hystrix AMQP messages and emits them as a Server Sent Event stream.

Deployment Details

> **Org**: `p-spring-cloud-services`
> **Space**: `instances`
> **CF Application Name**: `turbine-[guid]`, where `[guid]` is the service instance ID

Entry Points

GET /turbine.stream

This emits the Server Sent Event stream and is the only HTTP-based entry point. To authenticate the request, the Turbine application uses the Authorization header of the incoming request (containing the OAuth 2 Bearer token) to make a request to the CC API to check permissions on the service instance being accessed, as documented here. The service instance ID used in the permission check is supplied by an environment variable set at provision time by the Worker. This check is done once per SSE request, and because there is no session involved, it is not cached. However, SSE requests are kept alive until the browser window is closed, so the permission check will happen infrequently.

Hystrix AMQP messages

The Turbine application listens for AMQP messages posted by applications bound to the Circuit Breaker Dashboard service instance. Applications bound to the Circuit Breaker Dashboard receive credentials to the p-rabbitmq service instance in order to post these messages. All bound applications receive the same credentials.

External Dependencies

p-rabbitmq

As described earlier, bound applications use an instance of p-rabbitmq to post circuit breaker metrics. The Turbine application listens to these messages, aggregates the metrics, and emits a Server Sent Event stream. The credentials for the p-rabbitmq service instance are provided to the Turbine application though service instance binding. These same credentials are provided to applications bound to the Circuit Breaker Dashboard service instance, and would be visible to any user that is able to bind to the service instance. These credentials can be used for admin access to the RabbitMQ.

Since this RabbitMQ is only used for collecting and disseminating `@HystrixCommand` metrics from bound applications, one risk here is breaking the ability for metrics to be gathered and reported in the Hystrix dashboard. Doing this would not affect the operation of any application bound to the service instance.

The Cloud Controller

The Turbine application communicates to the CC via HTTPS to perform permission checks using the current user's token, as described in the section on the Server Sent Event stream. The URI to the CC is set by the Worker at provision time in the application's environment variables, under the name `CF_TARGET`. The environment variable can only be modified by the Operator.

Release Notes for Spring Cloud® Services on Pivotal Cloud Foundry

Release notes for Spring Cloud Services for Pivotal Cloud Foundry

Migrating from 1.3.x

For each client application, you must:

- Update the build file to use the new versions of the Spring Cloud Services client dependencies (see Client Dependencies)

Migrating from 1.2.x

For each client application, you must:

- Update the build file to use the new versions of the Spring Cloud Services client dependencies (see Client Dependencies)

Migrating from 1.1.x

For each client application, you must:

- Update the build file to use the new versions of the Spring Cloud Services client dependencies (see Client Dependencies)

Migrating from 1.0.x

For each client application, you must:

- Update the build file to use the new versions of the Spring Cloud Services client dependencies (see Client Dependencies)
- Update the build file to use the Maven BOM dependencies for Spring Boot and Spring Cloud (see Client Dependencies)

For each Circuit Breaker Dashboard service instance, you must:

- Upgrade the service instance (see Service Instance Upgrades), then unbind, rebind, and restart each of the service instance's client applications (this is due to a change in how service instance credentials are managed)

1.4.8

Release Date: 26th February 2018

Enhancements included in this release:

- The stemcell has been updated to 3468. This resolves the following issues:
  - [USN-3554-1]: curl vulnerabilities
  - [USN-3543-1]: rsync vulnerabilities
  - [USN-3547-1]: Libtasn1 vulnerabilities
  - [USN-3582-2]: Linux kernel (Xenial HWE) vulnerabilities

1.4.7

Release Date: 30th November 2017

Enhancements included in this release:

- The previous release used an incorrect version of the stemcell. This release corrects that.

1.4.6

Release Date: 15th November 2017

Fixes included in this release:

- The Config Server's Vault configuration now supports multiple types of Vault backends (it previously supported only `secret`).

1.4.5

Release Date: 2nd November 2017

Enhancements included in this release:

- The stemcell has been updated to the 3445 line of floating stemcells.

1.4.4

Release Date: 21st September 2017

Enhancements included in this release:

- The Spring Cloud Services service broker has been made more resilient in cases of the RabbitMQ for PCF cluster being unavailable.
- Troubleshooting of Spring Cloud Services service broker activity with the RabbitMQ for PCF service has been enhanced.

Fixes included in this release:

- Resolved an issue with Config Server's health check failing when the service instance was configured with a Git SSH backend.
- Resolved an issue where an upgrade of the Spring Cloud Services tile resulted in Config Server instances not retaining the `label` configuration parameter.

1.4.3

Release Date: 9th August 2017

Enhancements included in this release:

- The Spring Cloud Services service broker has been upgraded to Spring Cloud Dalston.SR2.

Fixes included in this release:

- Fixed a Config Server issue where Git repository passwords in configuration shown on the dashboard of a service instance using a composite backend were not masked.

1.4.2

Release Date: 31st July 2017

Enhancements included in this release:

- The Spring Cloud Services "Broker Deployer" lifecycle errand has been made more resilient in cases where the service broker worker's Cloud Foundry user or UAA client already exists.
- SSL certificate validation performed by the Spring Cloud Services "Broker Deployer" lifecycle errand is no longer case-sensitive.

Fixes included in this release:

- Fixed an issue with the "Broker Deployer" lifecycle errand where a missing route on the existing service broker application could cause the errand to fail.
- Fixed an issue with the "Broker Deployer" lifecycle errand where the presence of existing dependent service instances during an upgrade could cause the errand to fail.

1.4.1

Release Date: 7th July 2017

Enhancements included in this release:

- Messages sent by Circuit Breaker Dashboard to its RabbitMQ service instance now have a Time To Live (TTL) set, to prevent overloading of the RabbitMQ queue.
- The Spring Cloud Services service broker now can use a private domain to deploy service instance backing applications when no shared domains are available in the Elastic Runtime deployment. For more information, see the Domain Requirements section of the Prerequisites topic.

Fixes included in this release:

- Fixed a Config Server issue which could prevent backing application startup and resulted in backing application log and dashboard errors when the service instance was configured to connect to a Git repository via SSH.

Known issues in this release:

- The "Broker Registrar" lifecyle errand fails during the tile upgrade process if the Spring Cloud Services service broker's MySQL for PCF service instance was created under a plan name other than `100mb` (the default name for the 100 MB plan). If the service broker for your existing installation of Spring Cloud Services has a MySQL for PCF service instance with a service plan name other than `100mb` and the tile upgrade process has failed, follow the below steps to migrate the existing database to a `100mb` plan and continue the upgrade process.
  - In the MySQL for PCF tile configuration settings, create a service plan with name `100mb` and persistent disk of 100 MB.
  - Back up the existing broker database, following the steps in the Back Up Your MySQL for PCF v1.x Instance section of the Migrating a Database to MySQL for PCF v2 topic in the MySQL for PCF documentation.
  - Using the Cloud Foundry Command Line Interface tool (cf CLI), target the `system` org and the `p-spring-cloud-services` space:
    ```
    cf target -o system -s p-spring-cloud-services
    ```
  - Using the cf CLI, update the broker's MySQL for PCF service instance to the new `100mb` plan:
    ```
    cf update-service spring-cloud-broker-db -p 100mb
    ```
  - Re-run the Spring Cloud Services post-deploy errands.

1.4.0

Release Date: 29th May 2017

WARNING: If you are using Trusted System Certificates, do not install or upgrade to Spring Cloud Services 1.4.0. See “Known issues” below.

WARNING: If you have upgraded to Spring Cloud Services version 1.4.0 but have a Config Server service instance which accesses a Git repository via SSH and has not yet been upgraded, do not upgrade the service instance at this time. See “Known issues” below.

Enhancements included in this release:

- The Config Server now can use a HashiCorp Vault server as a configuration source, directly or through an HTTP or HTTPS proxy. For more information, see the Configuring with Vault topic in the Config Server documentation. In order to use the Vault support in a client application, you must update your application’s Spring Cloud Services client dependencies; see the Client Dependencies topic.
- The Config Server now can use a composite backend comprising one or more Git repositories and up to one HashiCorp Vault server. For more information, see the Composite Backends topic in the Config Server documentation.
- The Config Server’s `/health` endpoint, provided by Spring Boot Actuator, now displays failure information when the Config Server cannot use the provided configuration or otherwise encounters an error. For information about accessing this endpoint, see the Access Actuator Endpoints section of the The Service Broker and Instances topic.
- The Config Server’s backing application now logs failure information when the Config Server cannot use the provided configuration or otherwise encounters an error. For information about accessing the backing application logs, see the Read Backing Application Logs section of the The Service Broker and Instances topic.
- The Config Server dashboard now displays detailed failure information when the Config Server cannot use the provided configuration or otherwise encounters an error. For more information, see the Using the Dashboard topic in the Config Server documentation.
- The Spring Cloud Services service broker has been upgraded to Spring Boot 1.5.2 and Spring Cloud Dalston.RELEASE.

Known issues in this release:

- uaac commands used by Spring Cloud Services lifecycle errands fail if Pivotal Cloud Foundry® Operations Manager has been configured to use Trusted System Certificates (for example to support an internal root Certificate Authority). If you are using Trusted System Certificates, do not install or upgrade to this release. If you have already installed it following the tile installation process rather than the tile upgrade process, uninstall and delete the tile; if you have already upgraded to this release, please contact Pivotal Support for assistance.
- The “Broker Registrar” lifecyle errand fails during the tile upgrade process if the Spring Cloud Services service broker’s MySQL for PCF service instance was created under a plan name other than `100mb` (the default name for the 100 MB plan). If the service broker for your existing installation of Spring Cloud Services has a MySQL for PCF service instance with a service plan name other than `100mb` and the tile upgrade process has failed, follow the below steps to migrate the existing database to a `100mb` plan and continue the upgrade process.
  - In the MySQL for PCF tile configuration settings, create a service plan with name `100mb` and persistent disk of 100 MB.
  - Back up the existing broker database, following the steps in the Back Up Your MySQL for PCF v1.x Instance section of the Migrating a Database to MySQL for PCF v2 topic in the MySQL for PCF documentation.
  - Using the Cloud Foundry Command Line Interface tool (cf CLI), target the `system` org and the `p-spring-cloud-services` space:
    `cf target -o system -s p-spring-cloud-services`
  - Using the cf CLI, update the broker’s MySQL for PCF service instance to the new `100mb` plan:
    `cf update-service spring-cloud-broker-db -p 100mb`
  - Re-run the Spring Cloud Services post-deploy errands.
- A Config Server service instance which has a configuration source consisting of a Git repository reached via SSH may fail to start, will report a status of `DOWN` on its `health` endpoint (provided by Spring Boot Actuator), and will report on its dashboard that it “cannot initialize using the configuration that has been provided”. The backing application logs for the service instance may contain the phrases `reject HostKey` or `ssh protocol is not supported`.

  If you have upgraded Spring Cloud Services to version 1.4.0 but have a Config Server service instance which accesses a Git repository via SSH and has not yet been upgraded to the latest version, do not upgrade the service instance at this time. If you have such a service instance and it has been upgraded, follow the below steps to enable the Config Server to start (this will not correct the status given on the `health` endpoint or the dashboard message).
  - Locate the backing application for the service instance (see the Service Instance Management section of the The Service Broker and Instances topic).
  - Run `cf env [APPNAME]`, where `APPNAME` is the name of the backing application. From the output, copy the value of the `SPRING_APPLICATION_JSON` environment variable.
  - Within the `spring.cloud.config.server.git` object for the relevant Git configuration source, add a property `strictHostKeyChecking` with value `false`. Use `cf set-env` to set the new environment variable value:
    ```
    cf set-env config-xxx-xxx-xxx SPRING_APPLICATION_JSON '{"spring":{"cloud":{"config":{"server":{"git":
    {"strictHostKeyChecking":false, "uri:"...", "privateKey":"...", ...}'
    ```
  - Restage the backing application.
- Spring Cloud Services is affected by an issue in Spring Boot version 1.5.3 and is not compatible with that version. To be compatible with this release of Spring Cloud Services, client applications should use Spring Boot versions 1.5.0–1.5.2 or 1.5.4 and later.
- A Config Server service instance which has a configuration source consisting of a Git repository defined using placeholders (see the Placeholders in Repository URIs section of the Configuring with Git topic in the Config Server documentation) always clones that repository’s data to a location under the system temporary directory, which can be periodically emptied, causing a loss of cloned configuration data. By default, the Spring Cloud Services Config Server does not clone repository data under the temporary directory. To ensure that repository data is cloned elsewhere, do not use placeholders in a Git repository URI with this release.
- A Config Server service instance which has a configuration source consisting of a Git repository defined using the `label` setting will lose that setting in a tile upgrade from 1.3.x to 1.4.0.
- A client application using RabbitMQ and bound to a Circuit Breaker Dashboard service instance may fail to start up. This is due to an issue in the Spring Cloud Stream RabbitMQ binder. The issue is resolved in the binder’s version 1.2.1; this release of Spring Cloud Services depends on Spring Cloud OSS Dalston.RELEASE, which includes the binder’s version 1.2.0.

  If you wish to use RabbitMQ with a Circuit Breaker Dashboard service instance, you must manually override either the version of the Spring Cloud Stream RabbitMQ binder used by your application or the version of the Spring Cloud Maven BOM used by your application (the 1.2.1 release of the

binder is included in Spring Cloud OSS Dalston.SR1).

Using Gradle:

```
// Override managed version of Spring Cloud Stream RabbitMQ binder
compile ("org.springframework.cloud:spring-cloud-stream-binder-rabbit:1.2.1.RELEASE")

// Override version of OSS Spring Cloud
dependencyManagement {
   imports {
      mavenBom "org.springframework.cloud:spring-cloud-dependencies:Dalston.SR1"
   }
}
```

Using Maven:

```
<!-- Override managed version of Spring Cloud Stream RabbitMQ binder -->
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
   <version>1.2.1.RELEASE</version>
</dependency>

<!-- Override version of OSS Spring Cloud -->
<dependencyManagement>
  <dependencies>
     <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
        <version>Dalston.SR1</version>
        <type>pom</type>
        <scope>import</scope>
     </dependency>
  </dependencies>
</dependencyManagement>
```

Prerequisites to Installing Spring Cloud® Services for Pivotal Cloud Foundry

Page last updated:

Please ensure that your Pivotal Cloud Foundry (PCF) installation meets the below requirements before installing Spring Cloud Services.

Buildpack Requirements

Spring Cloud Services is built using Spring Boot 1.4, which requires version 3.8 or later of the Java Cloud Foundry buildpack. The default Java buildpack—the buildpack at the lowest position of all Java buildpacks—on your PCF installation must therefore be at version 3.8 or later.

You can use the Cloud Foundry Command Line Interface tool (cf CLI) to see the version of the Java buildpack that is currently installed.

```
$ cf buildpacks
Getting buildpacks...

buildpack              position  enabled  locked  filename
java_buildpack_offline  1         true     false   java-buildpack-offline-v3.8.1.zip
ruby_buildpack          2         true     false   ruby_buildpack-cached-v1.6.19.zip
nodejs_buildpack        3         true     false   nodejs_buildpack-cached-v1.5.15.zip
go_buildpack            4         true     false   go_buildpack-cached-v1.7.10.zip
```

If the default Java buildpack is older than version 3.8, you can download a newer version from Pivotal Network and update Pivotal Cloud Foundry by following the instructions in the Managing Custom Buildpacks topic. To ensure that the newer buildpack is the default Java buildpack, you may delete or disable the older buildpack or make sure that the newer buildpack is in a lower position.

If the default Java buildpack on the Pivotal Cloud Foundry platform is not at version 3.8 or later, you must specify an alternate buildpack that is at version 3.8 or later when installing the Spring Cloud Services product; see step 4 of the Installation topic.

Product Requirements

Spring Cloud Services requires the following Pivotal Cloud Foundry products to be installed:

- MySQL
- RabbitMQ

If they are not already installed, you can follow the steps listed in the Installation topic to install them along with Spring Cloud Services.

Important: If you enable the RabbitMQ® for Pivotal Cloud Foundry product's SSL support by providing it with SSL keys and certificates, you must enable the RabbitMQ product's TLS 1.0 support; otherwise, the Spring Cloud Services service broker will fail to create or update service instances. See the Configuring the RabbitMQ Service topic in the RabbitMQ for Pivotal Cloud Foundry documentation.

Security Requirements

You will need to update your Elastic Runtime SSL certificate as described in the Pivotal Cloud Foundry documentation. Generate one single certificate that includes all of the domains listed below, replacing `SYSTEM_DOMAIN.TLD` with your system domain and `APPLICATION_DOMAIN.TLD` with your application domain:

- `*.SYSTEM_DOMAIN.TLD`
- `*.APPLICATION_DOMAIN.TLD`
- `*.login.SYSTEM_DOMAIN.TLD`
- `*.uaa.SYSTEM_DOMAIN.TLD`

If any of these domains are not attributed to your Elastic Runtime SSL certificate, the installation of Spring Cloud Services will fail, and the installation logs will contain an error message that lists the missing domain entries:

```
Missing certs: *.uaa.example.com - exiting install. Please refer to the Security
Requirements section of the Spring Cloud Services prerequisites documentation.
```

Domain Requirements

If you are installing Spring Cloud Services version 1.4.0 or earlier, your Elastic Runtime deployment must have at least one shared domain available (see the Shared Domains section of the Routes and Domains topic in the PCF documentation). If the Elastic Runtime deployment does not have at least one shared domain available, the Spring Cloud Services service broker is unable to deploy service instance backing applications.

In Spring Cloud Services version 1.4.1 and later, service instance backing applications can be deployed using a private domain assigned to the `p-spring-cloud-services` org. If there is at least one shared domain available in the Elastic Runtime deployment, service instance backing applications will be deployed using the first shared domain available. If there is not at least one shared domain available in the Elastic Runtime deployment, service instance backing applications will be deployed using the first private domain assigned to the `p-spring-cloud-services` org.

For more information about the Spring Cloud Services service instance backing applications, see the Service Instances section of the The Service Broker and Instances topic.

Installing Spring Cloud® Services for Pivotal Cloud Foundry

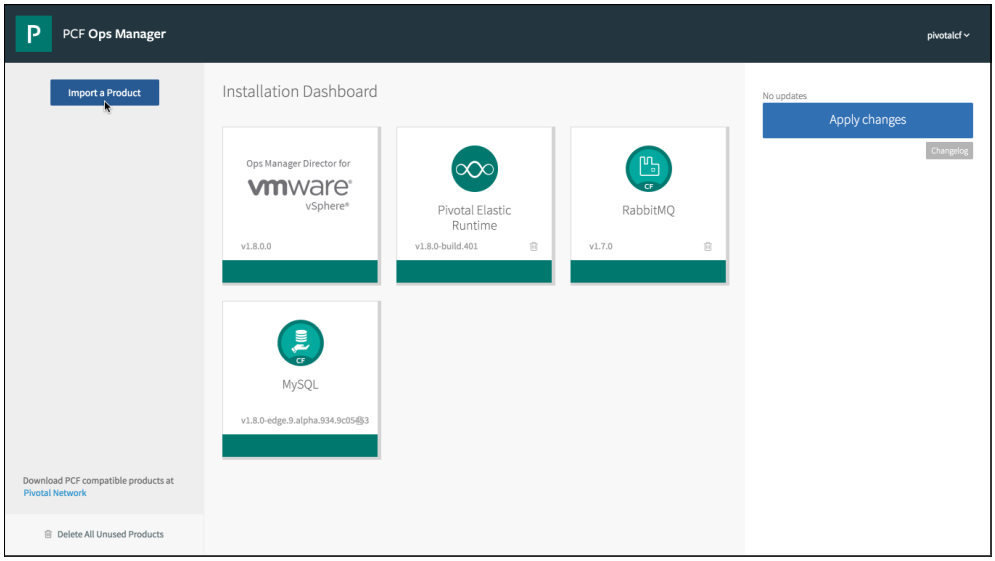
Page last updated:

Make sure that you have or have completed all products and requirements listed in the Prerequisites topic. Then follow the below steps to install Spring Cloud Services.

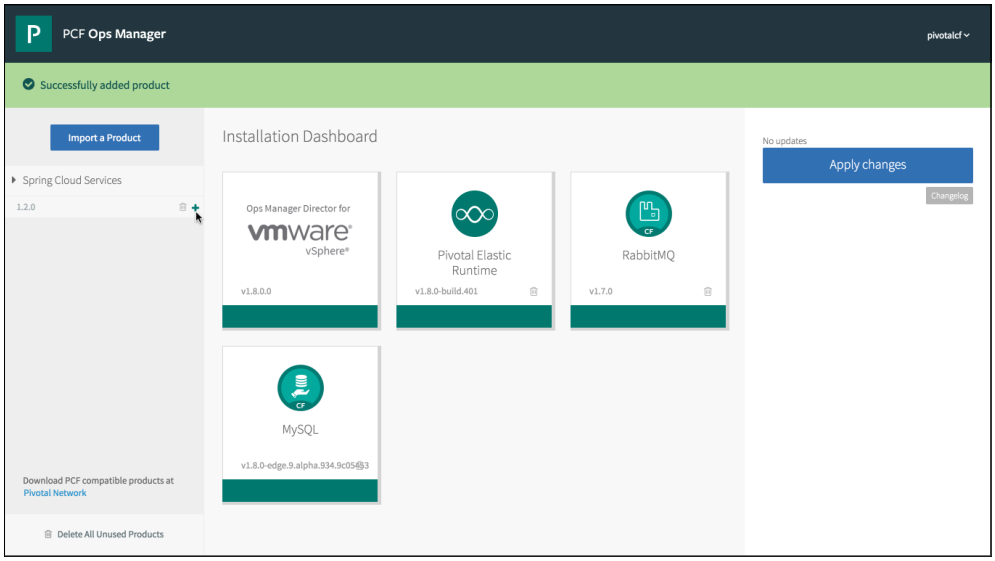
Installation Steps

1. Download Spring Cloud Services from Pivotal Network.

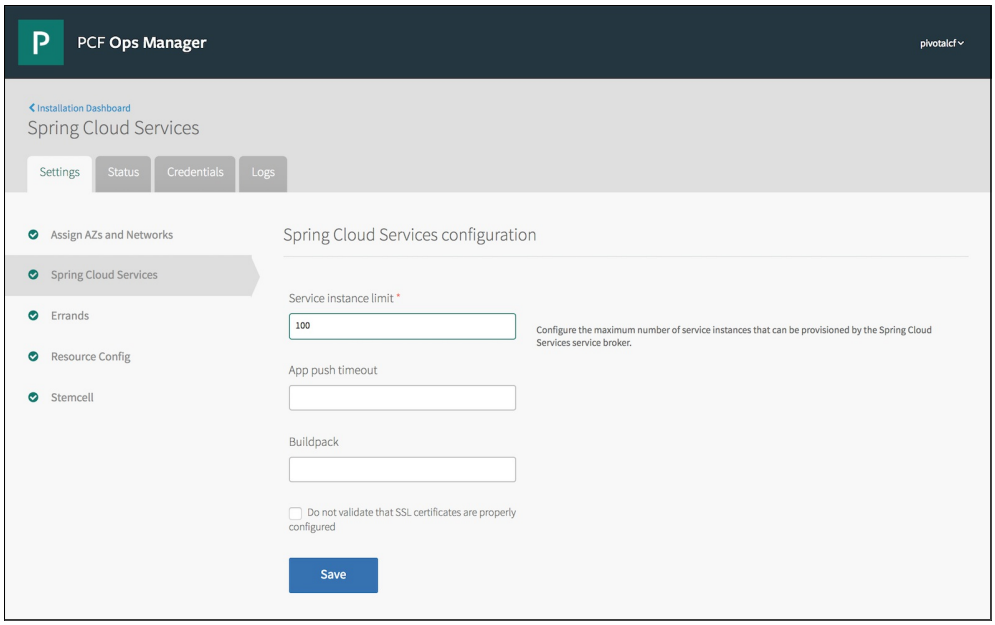
2. In the Installation Dashboard of Pivotal Cloud Foundry (PCF) Operations Manager, click **Import a Product** on the left sidebar to upload the Spring Cloud Services `.pivotal` file.



3. Hover over **Spring Cloud Services** in the **Available Products** list and click the **Add** » button.

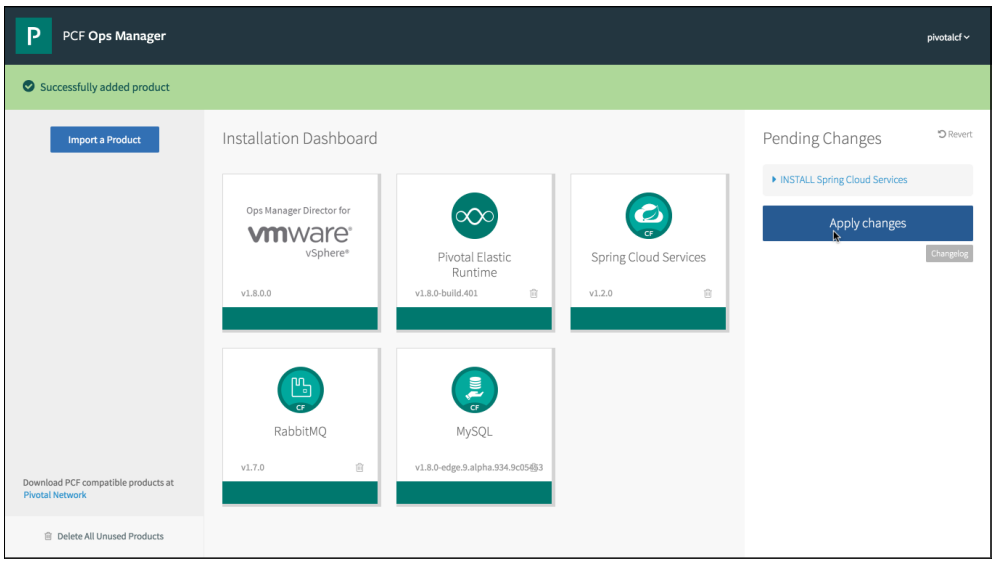


4. When the **Spring Cloud Services** tile appears in the **Installation Dashboard**, click it. In the **Settings** tab, click **Spring Cloud Services** to configure settings for Spring Cloud Services.

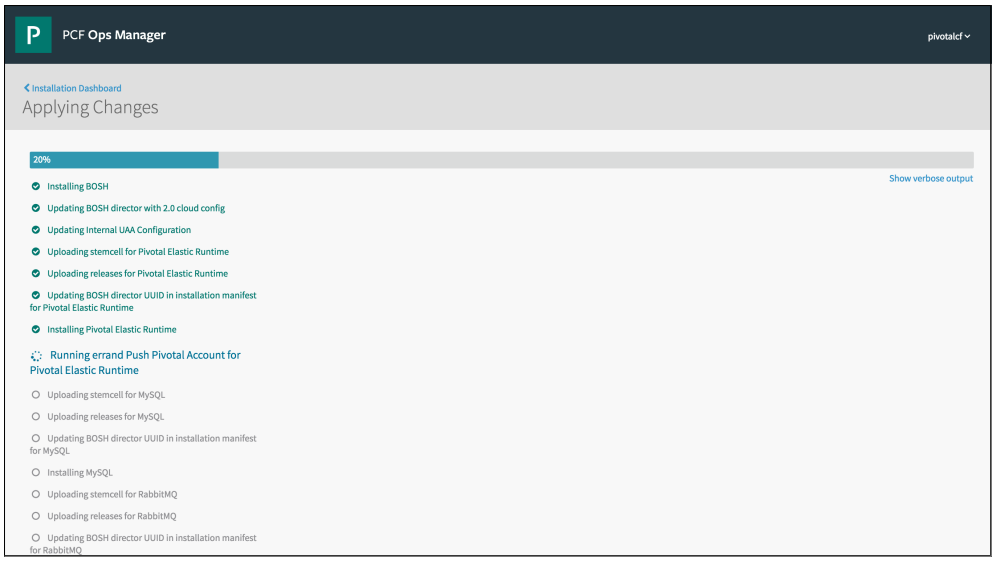


The **Service instance limit** setting sets the maximum number of service instances that the Spring Cloud Services service broker will allow to be provisioned (the default value is 100). This setting's value is also used to determine the memory quota for the org in which service instances are deployed; that org's quota will be equal to 1.5G times the configured service instance limit. The **App push timeout** setting is the number of minutes the broker allows for a service instance backing application to start. The **Buildpack** setting is the name of the Java buildpack that the Spring Cloud Services service broker will use to provision service instances (if this setting is left empty, the broker will use the default Java buildpack to provision service instances). The **Do not validate that SSL certificates are properly configured** checkbox, if checked, disables the validation that Spring Cloud Services performs by default on the Pivotal Cloud Foundry® Elastic Runtime SSL certificate.

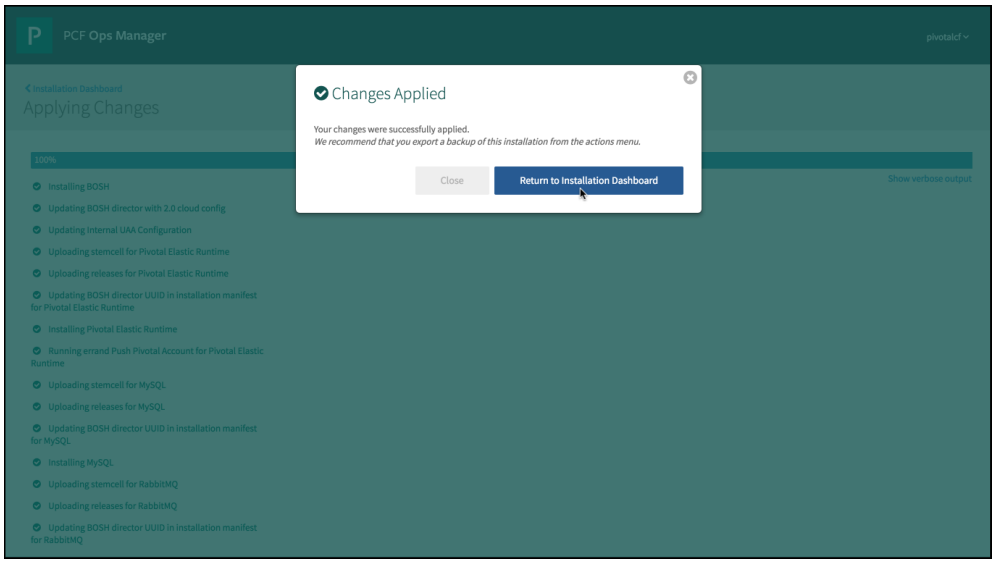
5. Return to the Installation Dashboard and start the installation process by clicking **Apply changes** on the right sidebar.



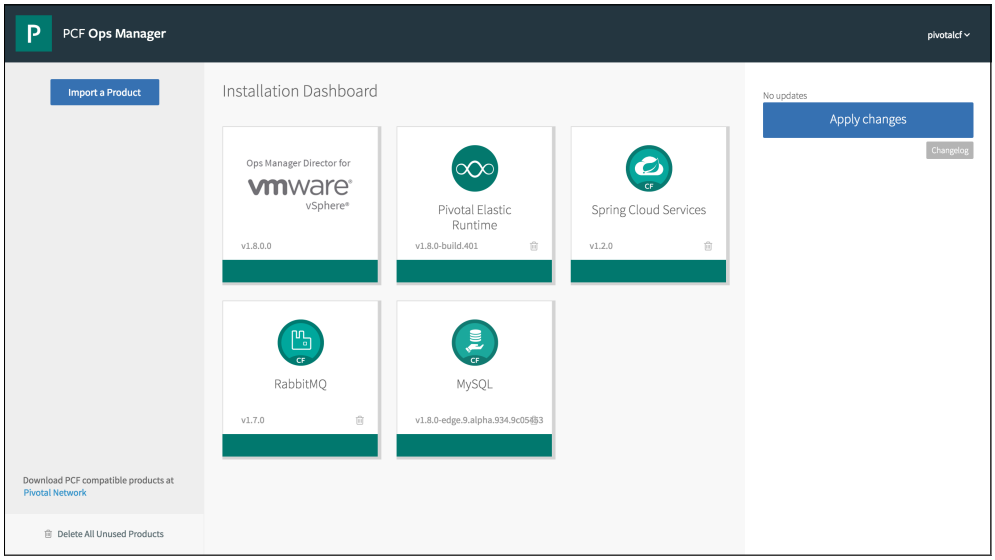
6. The installation process may take 20 to 30 minutes.



7. When the installation process is complete, you will see the dialog shown below. Click **Return to Installation Dashboard**.



8. Congratulations! You have successfully installed Spring Cloud Services.



Upgrading Spring Cloud® Services for Pivotal Cloud Foundry

Page last updated:

Product Upgradeability

Please see the relevant section below for important information regarding upgrades of the Spring Cloud Services product on your version of Pivotal Cloud Foundry.

Upgrades on Pivotal Cloud Foundry 1.6

When upgrading to a patch version of the Spring Cloud Services (SCS) product on Pivotal Cloud Foundry 1.6, you must consecutively apply all patch versions between your currently-installed SCS version and the version to which you wish to upgrade, and you may not skip a patch version. For example, you may not upgrade SCS 1.0.6 directly to SCS 1.0.8; instead, to upgrade SCS 1.0.6 to SCS 1.0.8, you must upgrade SCS 1.0.6 to SCS 1.0.7, then upgrade SCS 1.0.7 to SCS 1.0.8.

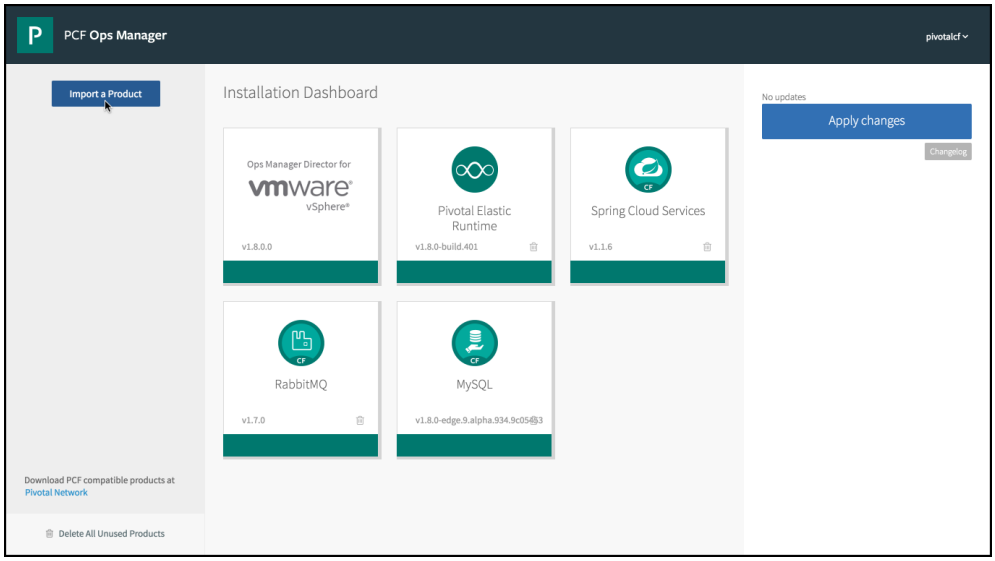
Upgrades on Pivotal Cloud Foundry 1.7

On Pivotal Cloud Foundry 1.7, you may not upgrade any version of the Spring Cloud Services product between 1.0.0 and 1.0.8 to any version older than 1.0.9 (e.g., you may not upgrade SCS 1.0.1 to SCS 1.0.2); however, you may upgrade any of these versions directly to 1.0.9 or later. You may upgrade any patch version of SCS after 1.0.9 directly to any later patch version and need not apply any intervening patch versions.

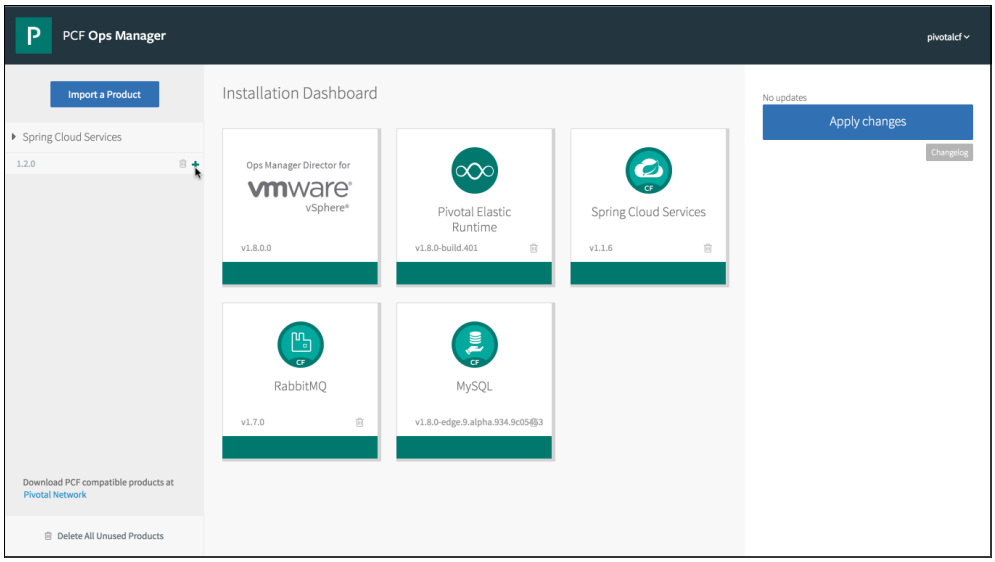
Product Upgrade Steps

Follow the below steps to upgrade the Spring Cloud Services product.

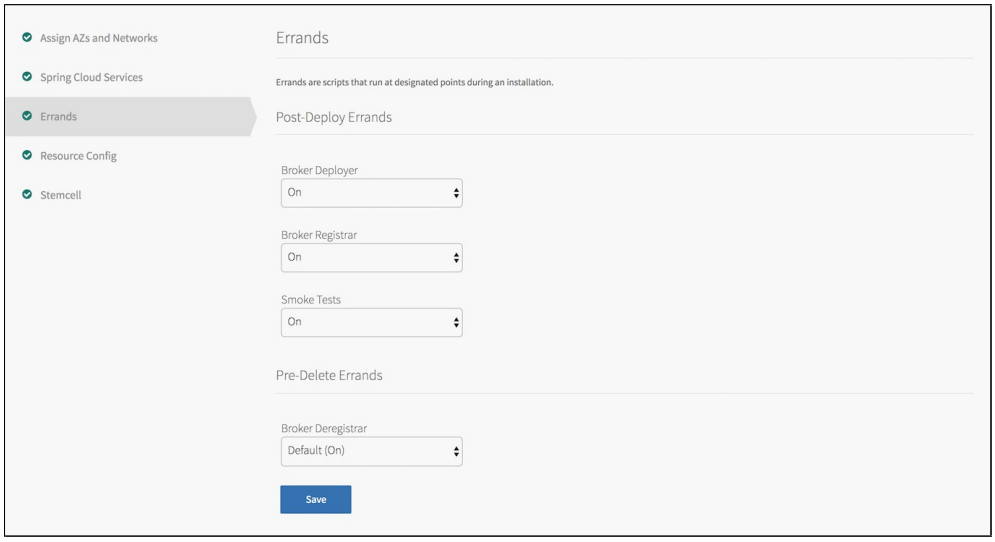
1. Download the latest version of Spring Cloud Services from Pivotal Network.
2. In the Installation Dashboard of Pivotal Cloud Foundry (PCF) Operations Manager, click **Import a Product** on the left sidebar to upload the `p-spring-cloud-services-<version>.pivotal` file.



3. After uploading the file, click the + button beside the version number under **Spring Cloud Services** in the product list.

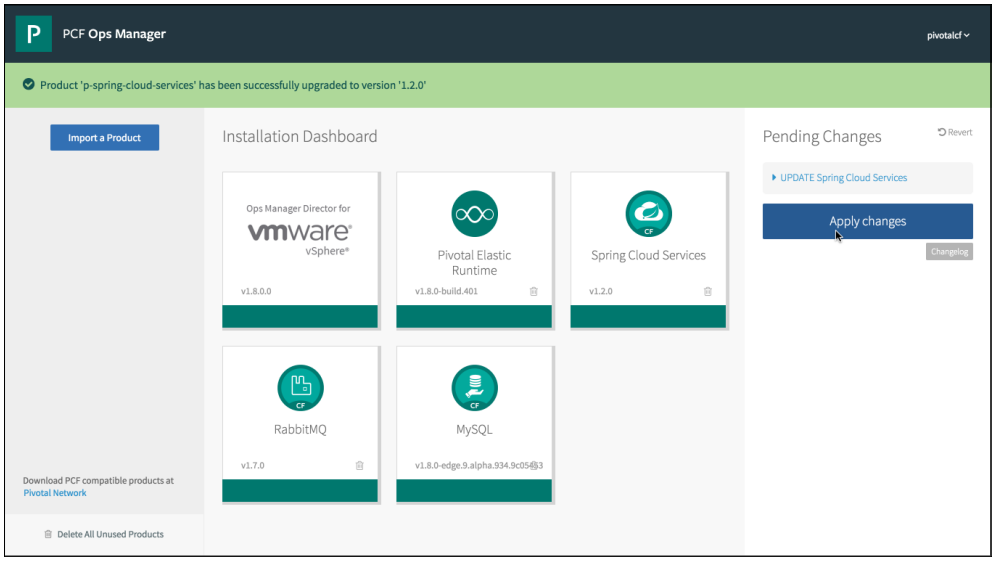


4. Click the Spring Cloud Services tile. In the **Errands** tab, ensure that all of the Spring Cloud Services errands are enabled.

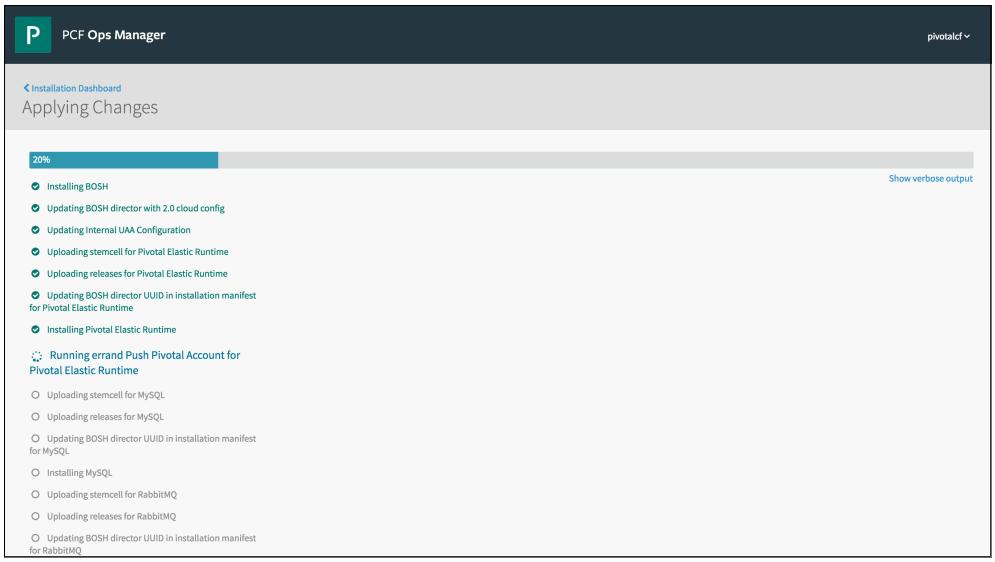


Important: The **Broker Registrar** post-deploy errand makes the Spring Cloud Services service broker's plans globally available (available to all orgs). This overrides any previous changes you have made to Spring Cloud Services service plan availability.

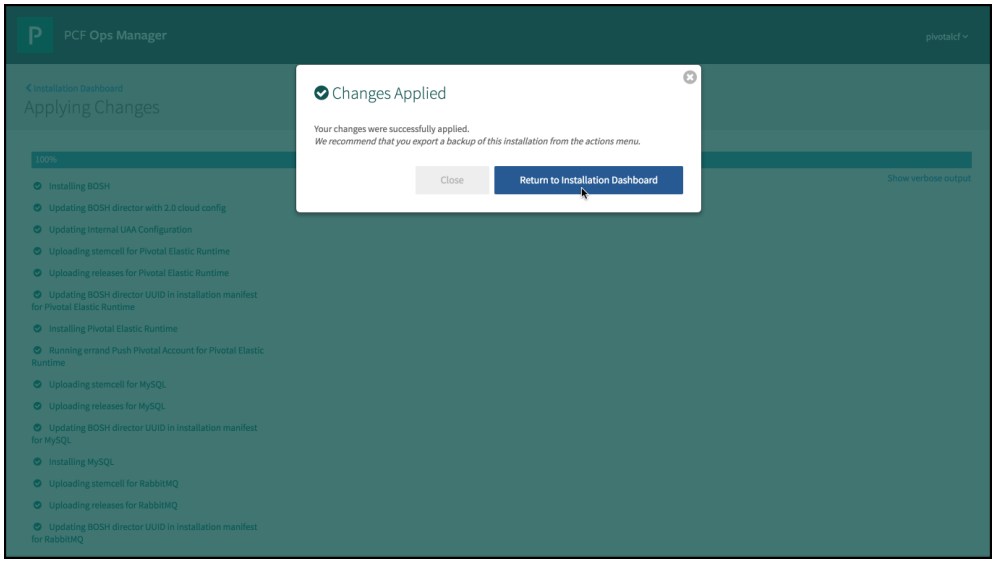
5. Return to the Installation Dashboard and click **Apply changes** on the right sidebar.



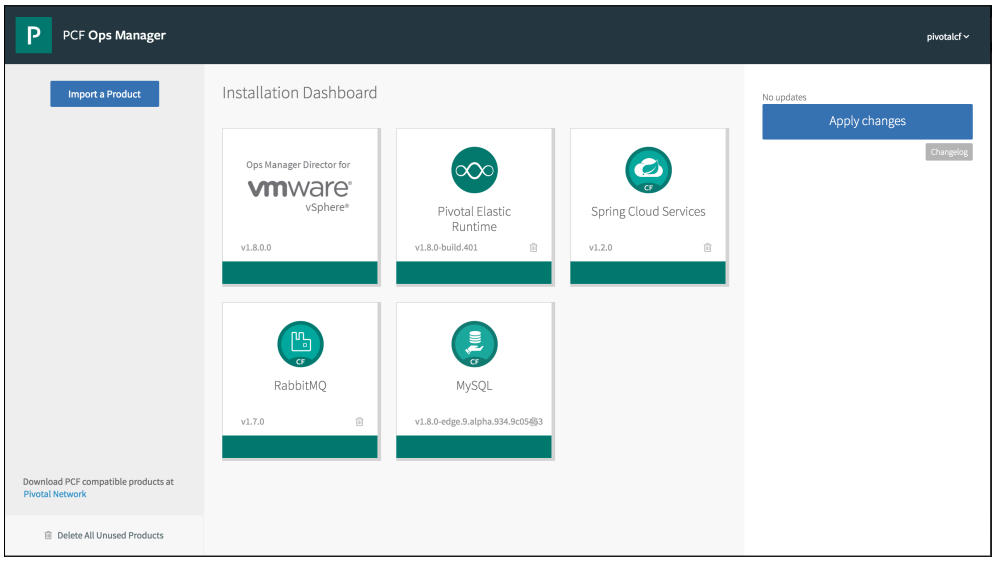
6. The upgrade process may take 20 to 30 minutes.



7. When the upgrade process is complete, you will see the dialog shown below. Click **Return to Installation Dashboard**.



8. Congratulations! You have successfully upgraded Spring Cloud Services.



Service Instance Upgrade Steps

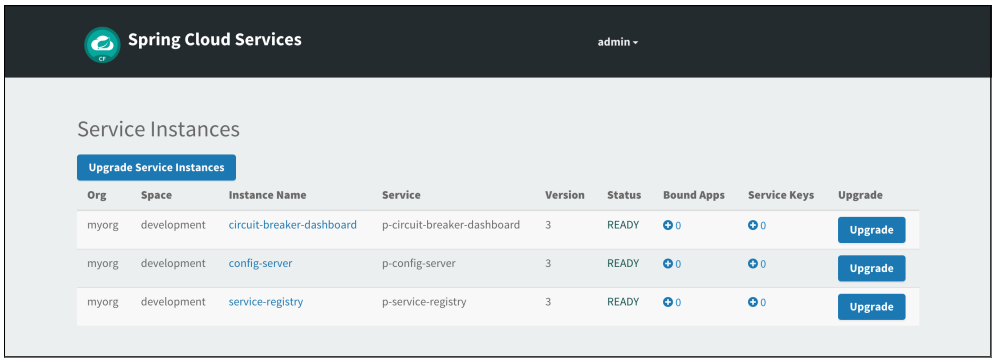
After upgrading the Spring Cloud Services product, you can follow the below steps to upgrade an individual service instance or to upgrade all service

instances. You can also use the Cloud Foundry Command Line Interface tool (cf CLI) to upgrade an individual service instance; see the Service Instance Upgrades topic for more information.

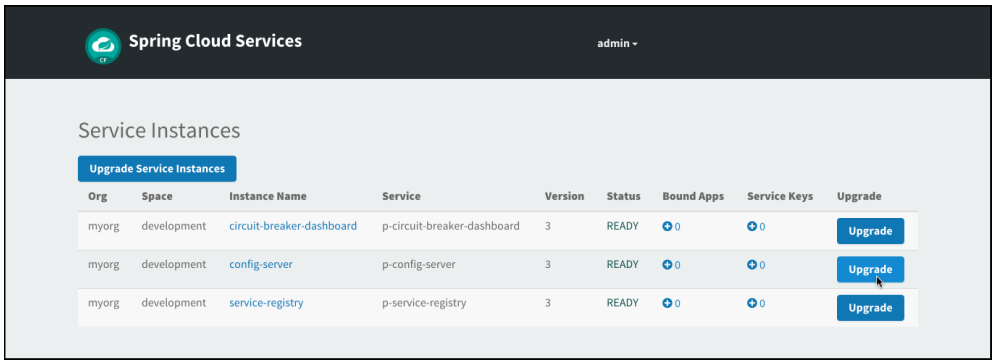
Important: If you have client applications which work with Spring Cloud Services 1.0.x service instances and are upgrading these service instances to 1.1.x, you must update your client applications to use the current version of the Spring Cloud Services client dependencies. See the Client Application Changes in 1.1.0 section of the Service Instance Upgrades topic.

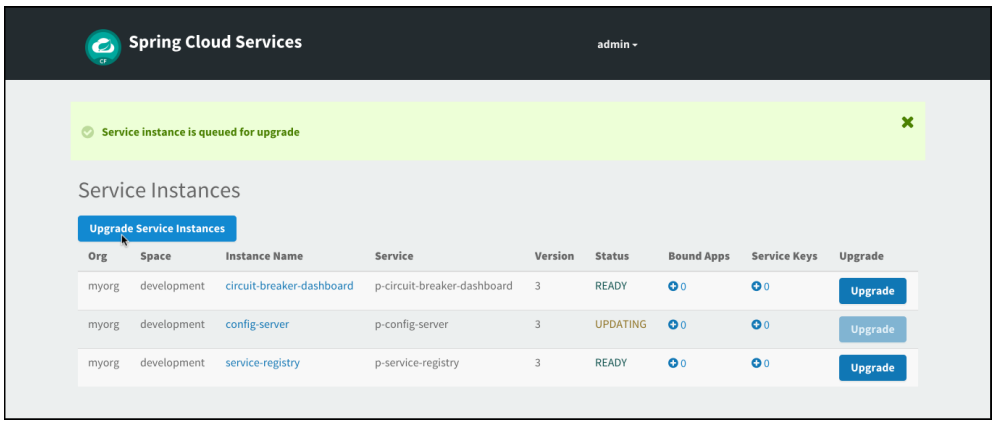
Important: After upgrading a Circuit Breaker Dashboard service instance from 1.0.x to 1.1.x, you must unbind, rebind, and restart any client applications which were bound to the instance. This is due to a change in how service instance credentials are managed.

1. To see the current version of a service instance, visit the Service Instances dashboard. (See the Service Instances Dashboard section of the The Service Broker and Instances topic.)

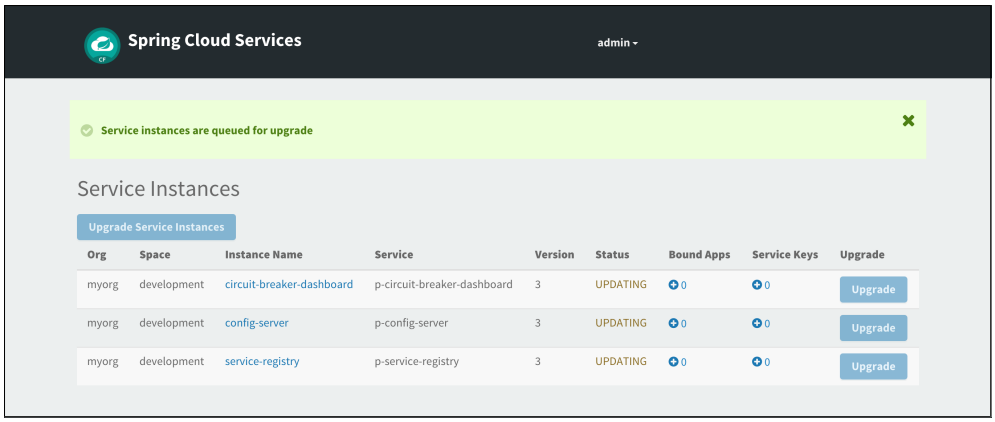


2. Click **Upgrade** in the listing for the service instance. To upgrade all service instances which are not at the versions included in the currently-installed Spring Cloud Services product, you may also click **Upgrade Service Instances**.

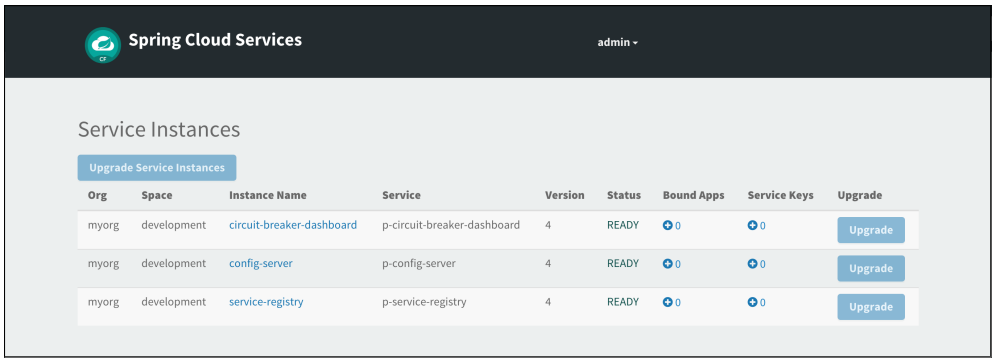




3. Wait while any backing applications belonging to the service instance or instances are upgraded. The dashboard will show the `UPDATING` status for an instance while it is being upgraded.



4. When the service instances have been upgraded, the dashboard will show the new versions.



Troubleshooting Spring Cloud® Services for Pivotal Cloud Foundry

See below for information about problems related to your Pivotal Cloud Foundry (PCF) platform configuration or Spring Cloud Services or other product installation.

Installation: “java.lang.NoClassDefFoundError: org/springframework/context/ApplicationContextInitializer”

If your installation of Spring Cloud Services fails and the `spring-cloud-broker` application logs (see the Read Broker Application Logs section of the The Service Broker and Instances topic for information about viewing those logs) include the following:

```
Caused by: java.lang.NoClassDefFoundError:
org/springframework/context/ApplicationContextInitializer
```

You may need to update the Java buildpack on your PCF deployment. Spring Cloud Services 1.3.x is based on Spring Boot 1.4 and is compatible with the Cloud Foundry Java buildpack version 3.8 or later. If the Java buildpack on your PCF deployment is older than 3.8, update it and continue the installation of Spring Cloud Services.

“java.util.concurrent.TimeoutException: Execution took longer than 180000 MILLISECONDS”

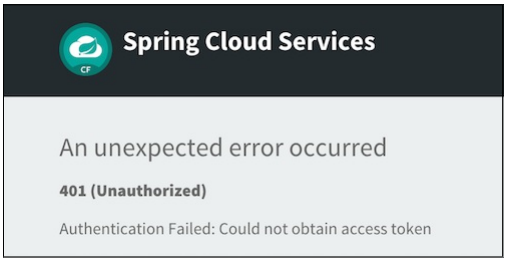
If the broker fails to provision a service instance and the logs for the `spring-cloud-broker-worker` application (see the Read Broker Application Logs section of the The Service Broker and Instances topic) contain the following:

```
java.util.concurrent.TimeoutException: Execution took longer than 180000 MILLISECONDS
```

This is due to a problem encountered during application deployments to your PCF deployment. Verify that you can deploy applications without issues to your PCF deployment (see “Starting or staging an application results in an InsufficientResources error” in the PCF Knowledge Base).

Service Registry: “Authentication Failed: Could not obtain access token”

If you encounter a message similar to the following when attempting to access a Service Registry dashboard:



with the following text:

> **401 (Unauthorized)**
>
> Authentication Failed: Could not obtain access token

If your PCF deployment is using a self-signed certificate, ensure that it includes all of the domains listed in the Security Requirements section of the Prerequisites topic. (See the Providing a Certificate for your SSL Termination Point topic in the Pivotal Cloud Foundry documentation for more information about configuring the certificate.) If you update the certificate, you will need to either restart the service instance’s backing application(s) or recreate the service instance, or you may continue to see this message and be unable to access the Service Registry dashboard.

“No subject alternative DNS name matching p-spring-cloud-services.uaa.example.com found”

If you encounter an exception message similar to the following:

```
org.springframework.web.client.ResourceAccessException: I/O error on POST request for
"https://p-spring-cloud-services.uaa.example.com/oauth/token":
java.security.cert.CertificateException: No subject alternative DNS name matching
p-spring-cloud-services.uaa.example.com found.; nested exception is
javax.net.ssl.SSLHandshakeException: java.security.cert.CertificateException: No
subject alternative DNS name matching p-spring-cloud-services.uaa.example.com found.
```

Ensure that your PCF installation meets the requirements described in the Security Requirements section of the Prerequisites topic. Be sure that your Elastic Runtime SSL certificate includes all wildcards listed in that section and has a separate wildcard for each subdomain.

“javax.net.ssl.SSLException: Received fatal alert: protocol_version”

The Cloud Foundry Command Line Interface tool (cf CLI) `create-service` command may return an error similar to the following:

```
Server error, status code: 502, error code: 10001, message: Service broker error:
javax.net.ssl.SSLException: Received fatal alert: protocol_version
```

Spring Cloud Services requires the RabbitMQ for PCF product. If you have provided SSL certificates and keys to RabbitMQ for PCF, you must also enable its TLS 1.0 support in order to use it with Spring Cloud Services. See the note at the end of the Product Requirements section of the Prerequisites topic.

“com.rabbitmq.client.AuthenticationFailureException: ACCESS_REFUSED - Login was refused using authentication mechanism PLAIN”

If you encounter the following exception message in Spring Cloud Services broker logs:

```
org.springframework.amqp.AmqpAuthenticationException:
com.rabbitmq.client.AuthenticationFailureException: ACCESS_REFUSED - Login was
refused using authentication mechanism PLAIN. For details see the broker
logfile.
```

The credentials used by the Spring Cloud Services broker to access its RabbitMQ for PCF service instance may be invalid. This can happen after restoring a backup of Pivotal Cloud Foundry® Elastic Runtime, as a restored RabbitMQ for PCF service instance may not accept the restored credentials used by the Spring Cloud Services broker. You can update the Spring Cloud Services broker’s credentials by unbinding the RabbitMQ service instance from the broker applications (the `spring-cloud-broker` and `spring-cloud-broker-worker` applications) and then rebinding.

To unbind the RabbitMQ service instance from the Spring Cloud Services broker applications and rebind it, run the following cf CLI commands:

```
$ cf target -o system -s p-spring-cloud-services
$ cf unbind-service spring-cloud-broker spring-cloud-broker-rmq
$ cf unbind-service spring-cloud-broker-worker spring-cloud-broker-rmq
$ cf bind-service spring-cloud-broker spring-cloud-broker-rmq
$ cf restage spring-cloud-broker
$ cf bind-service spring-cloud-broker-worker spring-cloud-broker-rmq
$ cf restage spring-cloud-broker-worker
```

If the PCF environment is newly restored and the RabbitMQ service instance’s queues are empty, you may also unbind the RabbitMQ service instance from the broker applications, delete it, create a new instance, and then bind that instance to the Spring Cloud Services broker applications.

WARNING: Deleting and replacing the RabbitMQ service instance will destroy the original instance’s virtual host and any data in the instance’s queues. If the RabbitMQ service instance contains messages and is on a production system, unbind and rebind it instead.

To delete and replace the RabbitMQ service instance used by the Spring Cloud Services broker applications, run the following cf CLI commands:

```
$ cf target -o system -s p-spring-cloud-services
$ cf unbind-service spring-cloud-broker spring-cloud-broker-rmq
$ cf unbind-service spring-cloud-broker-worker spring-cloud-broker-rmq
$ cf delete-service -f spring-cloud-broker-rmq
$ cf create-service p-rabbitmq standard spring-cloud-broker-rmq
$ cf bind-service spring-cloud-broker spring-cloud-broker-rmq
$ cf restage spring-cloud-broker
$ cf bind-service spring-cloud-broker-worker spring-cloud-broker-rmq
$ cf restage spring-cloud-broker-worker
```

Tile Configuration

Page last updated:

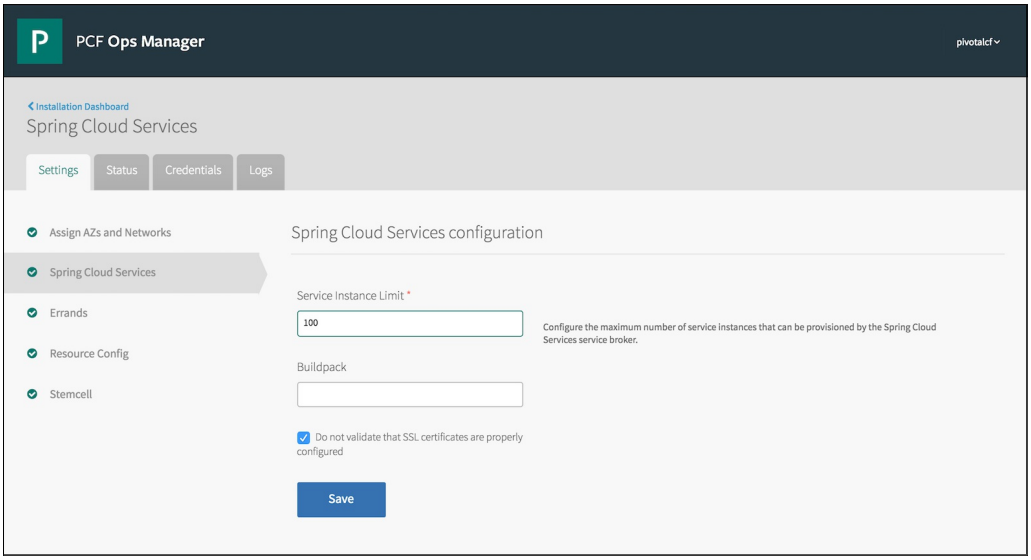
See below for information about configuration options and capacity requirements of the Spring Cloud® Services tile.

Tile Configuration Options

The Spring Cloud Services tile includes settings for various options. You can configure these by visiting the Installation Dashboard of Pivotal Cloud Foundry® Operations Manager, clicking the Spring Cloud Services tile, and then navigating to the **Spring Cloud Services** pane.

Configure Service Instance Limit

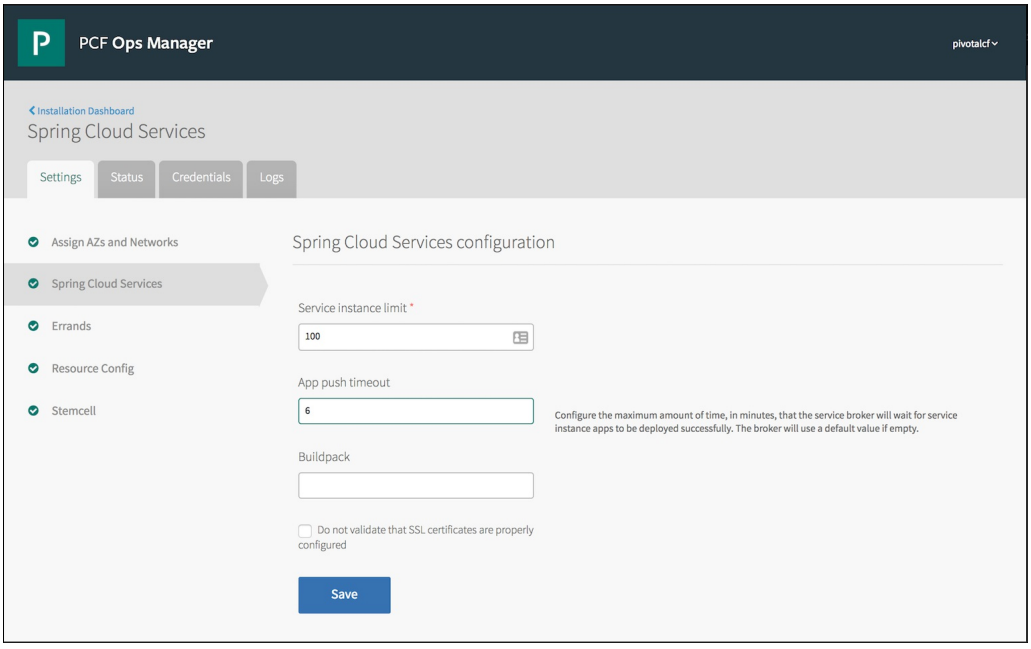
By default, the Spring Cloud Services broker will only provision up to 100 service instances. This service instance limit is also used to determine the memory quota for the org in which service instance backing applications are deployed; the org's memory quota will be equal to 1.5G times the service instance limit. You can configure this limit in the Spring Cloud Services tile settings.



In the **Service Instance Limit** field, enter the new service instance limit. Click **Save**, return to the Installation Dashboard, and apply your changes. The broker now can provision up to as many service instances as you specified.

Set Push Timeout for Backing Applications

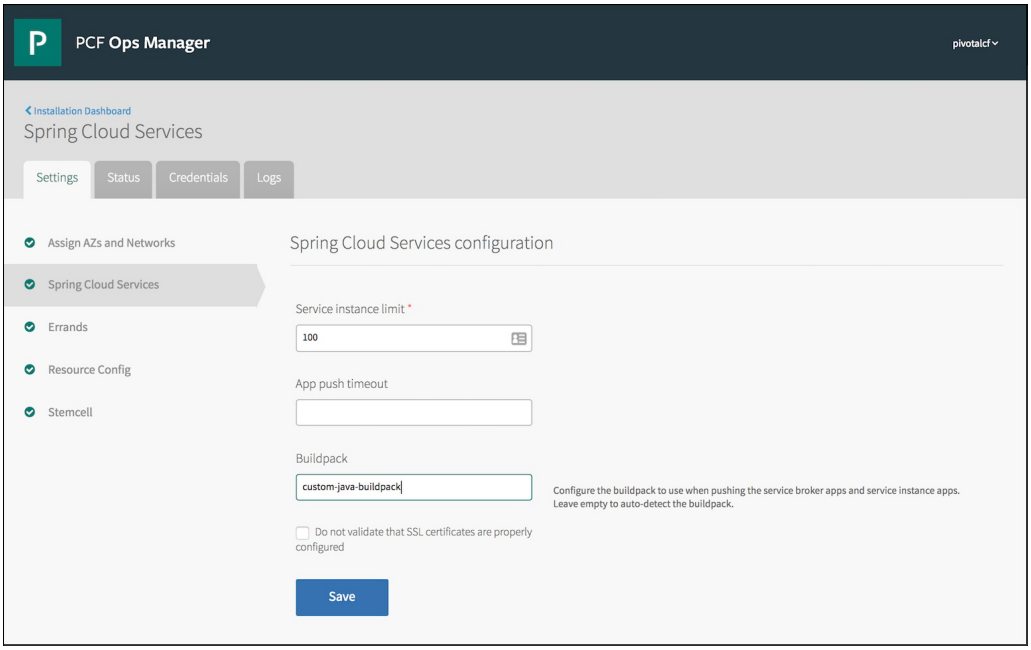
By default, the Spring Cloud Services service broker allows four (4) minutes for a service instance backing application to be staged and start. To use another timeout interval, you can specify a different number of minutes in the Spring Cloud Services tile settings.



In the **App push timeout** field, enter the desired number of minutes. Click **Save**, return to the Installation Dashboard, and apply the change. The broker will now allow the specified number of minutes for service instance backing applications to start.

Set Buildpack for Service Instances

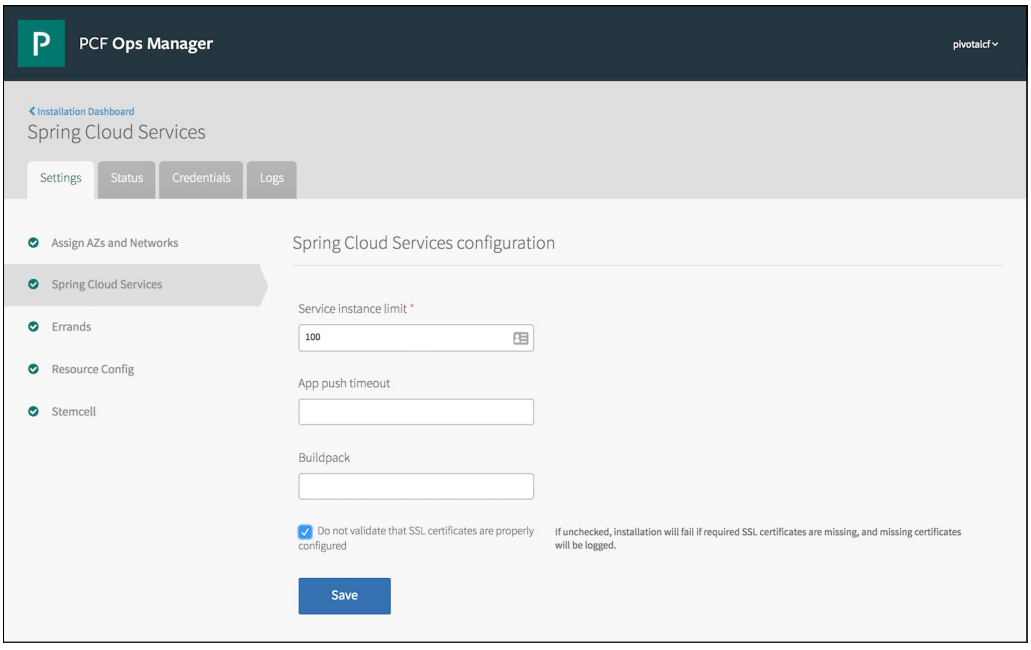
By default, the Spring Cloud Services service broker stages service instance backing applications using the buildpack chosen by the platform's buildpack detection; normally, this will be the highest-priority Java buildpack. To cause the broker to use a particular Java buildpack regardless of priority, you can set the name of the buildpack to use in the Spring Cloud Services tile settings.



In the **Buildpack** field, enter the name of the desired Java buildpack. Click **Save**, return to the Installation Dashboard, and apply your changes. The broker will now use the selected buildpack to stage service instance backing applications.

Skip SSL Certificate Validation

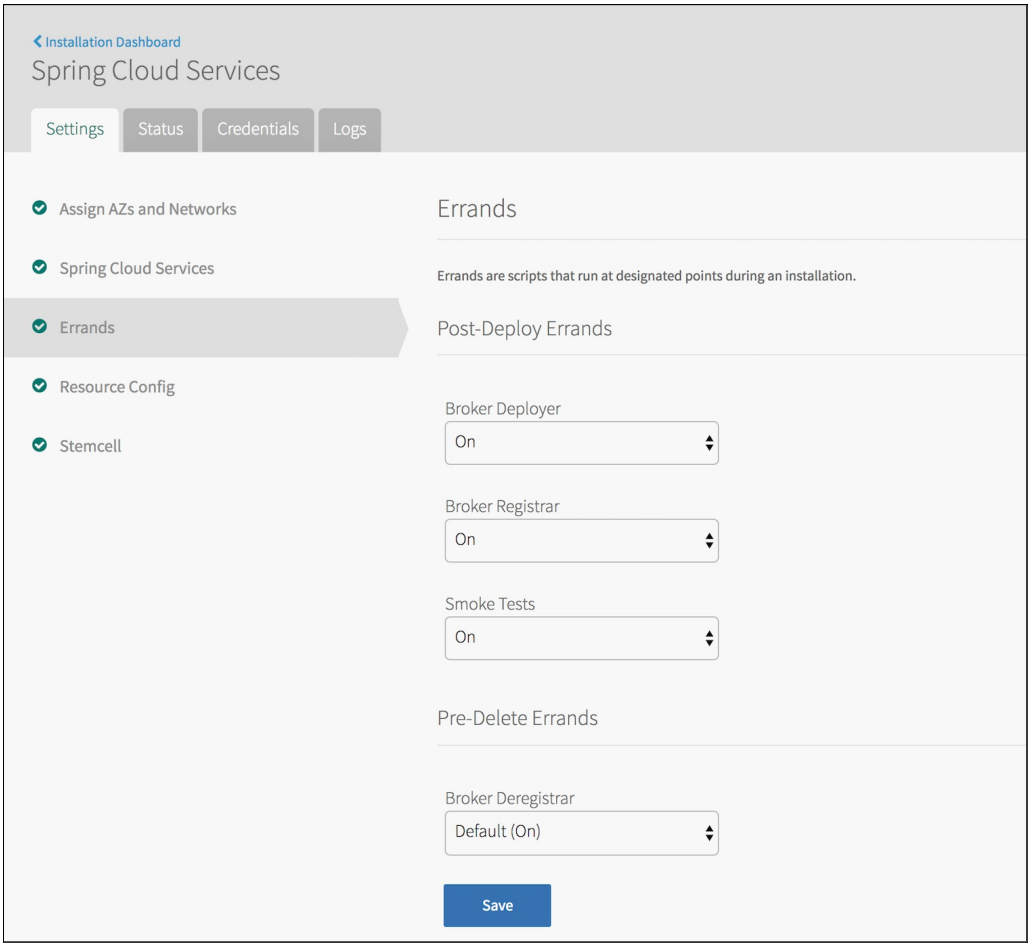
By default, Spring Cloud Services checks the Pivotal Cloud Foundry® Elastic Runtime SSL certificate for the domains included in the Security Requirements section of the Prerequisites topic. You can disable this validation by checking the **Do not validate that SSL certificates are properly configured** checkbox.



Click **Save** and return to the Installation Dashboard, then apply your changes. Spring Cloud Services will not validate the Elastic Runtime certificate as part of its installation process.

Errand Configuration

The Spring Cloud Services tile includes four lifecycle errands. You can view these by visiting the Installation Dashboard of Pivotal Cloud Foundry® Operations Manager, clicking the Spring Cloud Services tile, and then navigating to the **Errands** pane.



Spring Cloud Services includes three post-deploy errands and one pre-delete errand. These errands can be individually set to run conditionally (**When Changed**), to always run (**On**), or to never run (**Off**).

Important: To ensure that your PCF installation receives important updates to the tile, Pivotal recommends that all Spring Cloud Services lifecycle errands be set to **On**.

In the post-deploy errands, **Broker Deployer** pushes the Spring Cloud Services service broker and worker applications to PCF. **Broker Registrar** registers the service broker with the Cloud Controller and makes its service plans available to all orgs. **Smoke Tests** runs several tests to ensure service availability and correct configuration. The pre-delete errand, **Broker Deregistrar**, deletes all orgs and spaces specific to Spring Cloud Services and deregisters the broker from the Cloud Controller.

Capacity Requirements

Below are usage requirements of the applications backing a single service instance (SI) of each Spring Cloud Services service type.

Service Type	Memory Limit / SI	Disk Limit / SI
Config Server	1 GB	1 GB
Service Registry	1 GB	1 GB
Circuit Breaker Dashboard	2 GB	2 GB

Spring Cloud Services service types may also be backed by non-SCS service instances. For example, a Circuit Breaker Dashboard service instance uses a RabbitMQ for Pivotal Cloud Foundry service instance for communication between its two backing applications.

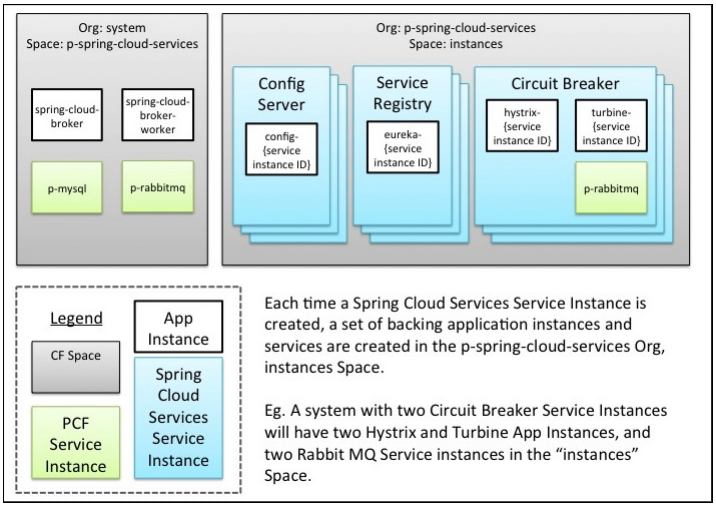
The Service Broker and Instances

Page last updated:

See below for information about Spring Cloud® Services' deployment model and other information which may be useful in operating its services or client applications.

Service Orchestration

Spring Cloud Services provides a series of Managed Services on Pivotal Cloud Foundry (PCF). It uses Cloud Foundry's Service Broker API to manage the three services—Config Server, Service Registry, and Circuit Breaker Dashboard—that it makes available. See below for information about Spring Cloud Services's broker implementation.



The Service Broker

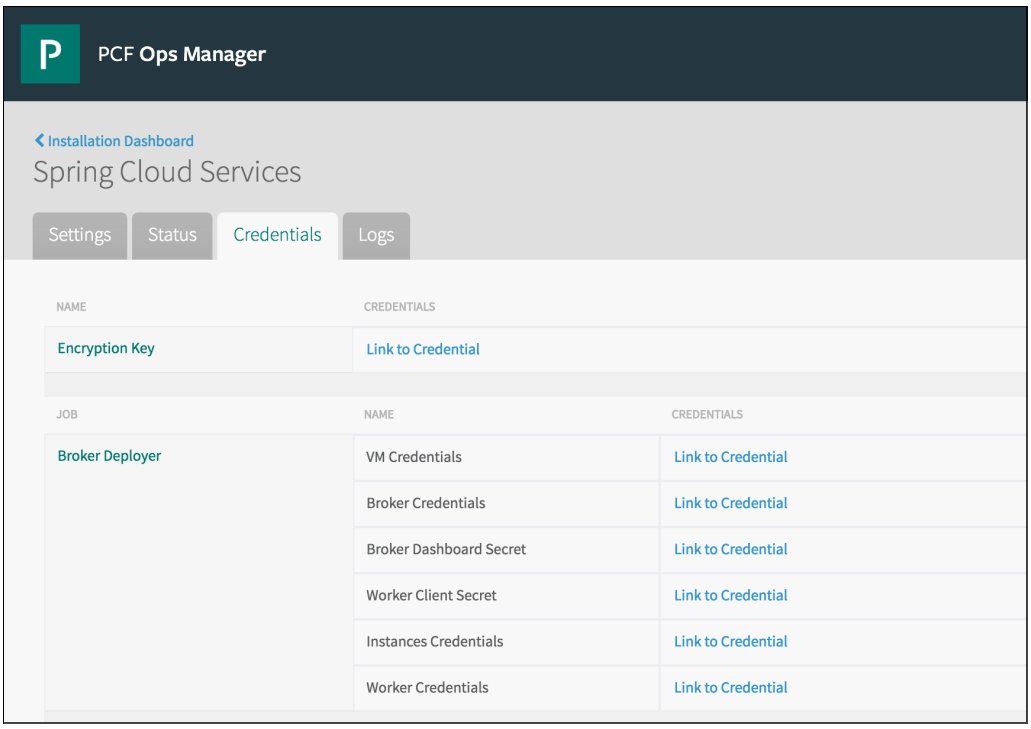
The service broker's functionality is divided between the following two Spring Boot applications, which are deployed in the "system" org to the "p-spring-cloud-services" space.

- spring-cloud-broker ("the SB"): Implements the Service Broker API.
- spring-cloud-broker-worker ("the Worker"): Acts on provision, deprovision, bind, and unbind requests.

The broker relies on two other Pivotal Cloud Foundry products, MySQL for Pivotal Cloud Foundry and RabbitMQ® for Pivotal Cloud Foundry, for the following service instances.

- **spring-cloud-broker-db**: A MySQL database used by the SB.
- **spring-cloud-broker-rmq**: A RabbitMQ queue used for communication between the SB and the Worker.

You can obtain the broker username and password from the Spring Cloud Services tile in Pivotal Cloud Foundry® Operations Manager. Click the Spring Cloud Services tile, and in the **Credentials** tab, follow the link to the Broker Credentials.



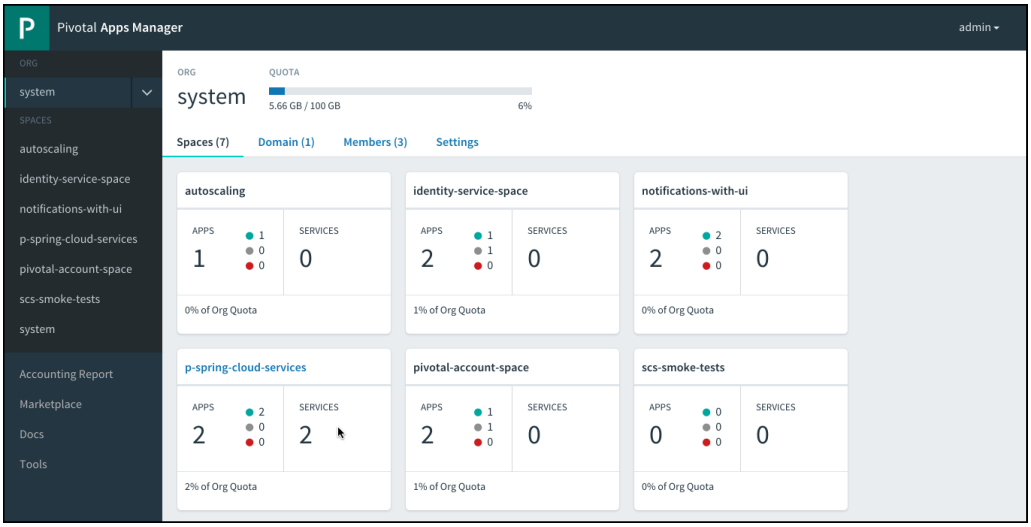
Broker Upgrades

The Spring Cloud Services product upgrade process checks before redeploying the service broker to see whether the broker applications' version has changed. If the version has not changed, the upgrade process will continue without redeploying either the SB or the Worker.

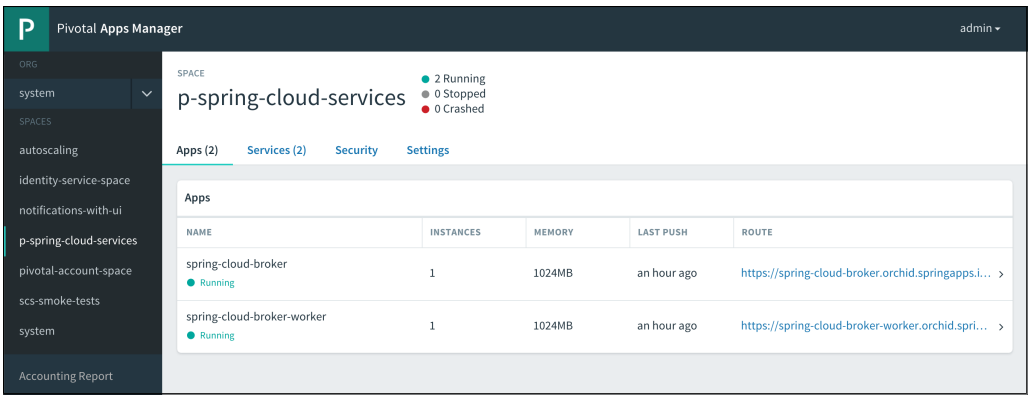
The SB application and the Worker application are deployed using a blue-green deployment strategy. During an upgrade of the service broker, the broker will continue processing requests to provision, deprovision, bind, and unbind service instances, without downtime.

Access Broker Applications in Apps Manager

To view the broker applications in Pivotal Cloud Foundry® Apps Manager, log into Apps Manager as an admin user and select the "system" org.



The applications are deployed in the "p-spring-cloud-services" space.



Service Instance Management

A service instance is backed by one or more Spring Boot applications deployed by the Worker in the “p-spring-cloud-services” org to the “instances” space.

A service instance is assigned a GUID at provision time. Backing applications for services include the GUID in their names:

Config Server

- config-[GUID]: A Spring Cloud Config server application.

Service Registry

- eureka-[GUID]: A Spring Cloud Netflix Eureka server application.

Circuit Breaker Dashboard

- hystrix-[GUID]: A Spring Cloud Netflix Hystrix server application.
- turbine-[GUID]: A Spring Cloud Netflix Turbine application.
- **rabbitmq-[GUID]**: A RabbitMQ for PCF service instance.

Access Service Instance Backing Applications in Apps Manager

To view a backing application for a service instance in Pivotal Cloud Foundry® Apps Manager, get the service instance’s GUID and look for the corresponding application in the “instances” space.

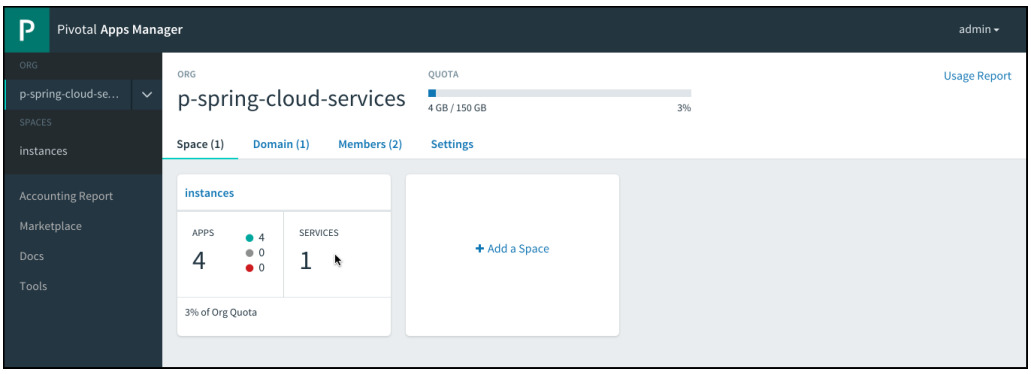
Target the org and space of the service instance.

```
$ cf target -o myorg -s development

API endpoint:   https://api.cf.wise.com (API version: 2.43.0)
User:           admin
Org:            myorg
Space:          development

$ cf services
Getting services in org myorg / space development as admin...
OK

name            service           plan       bound apps   last operation
config-server   p-config-server   standard   cook         update succeeded
```

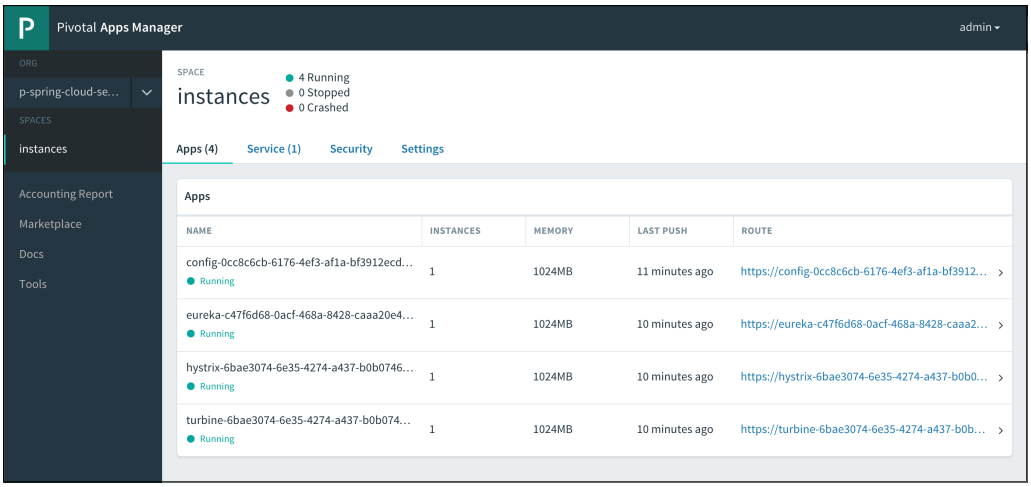
Run `cf service` with the `CF_TRACE` environment variable set to `true`. Copy the value of `resources.metadata.guid`, which is the service instance GUID.

```
$ CF_TRACE=true cf service config-server

REQUEST: [2016-06-27T20:45:24-05:00]
[...]

RESPONSE: [2016-06-27T20:45:25-05:00]
[...]

{
  "total_results": 1,
  "total_pages": 1,
  "prev_url": null,
  "next_url": null,
  "resources": [
    {
      "metadata": {
        "guid": "51711835-4626-4823-b5a1-e5d91012f3f2",
```

Log into Apps Manager as an admin user and select the “p-spring-cloud-services” org. The applications are deployed in the “instances” space.



Look for the backing application named as described above, with the prefix specific to the service type and the service instance GUID.



Get Service Instance Information

Each backing application for a service instance has the Spring Boot Actuator `info` endpoint enabled and accessible at `/info`. You can obtain version and other information about a service instance by accessing this endpoint.

Locate a backing application for a service instance as described in the Access Service Instance Backing Applications in Apps Manager section. In Apps Manager, on the page for the backing application, click the **Route** tab and copy the route for the application.

Append `/info` to the route for the application. In the example above, the resulting URI would be:

```
https://config-021b3d73-3ffc-4cb8-a105-26f5aaa54dff.wise.com/info
```

Visit the URI in a browser or request it from another client (e.g. curl at a command line). The response is JSON, as in the following example for a Config Server service instance:

```
{
  "version":"5",
  "git":{
    "commit":{
      "time":1478639430000,
      "id":"6e139ee"
    },
    "branch":"HEAD"
  }
}
```

The response may include additional fields depending on the service type, as in the following example for a Service Registry service instance:

```
{
 "issuer": "https://p-spring-cloud-services.uaa.wise.com/oauth/token",
 "version": "5",
 "nodeCount": "1",
 "git": {
  "commit": {
   "time": 1478639430000,
   "id": "6e139ee"
  },
  "branch": "HEAD"
 },
 "peers": []
}
```


In addition to the version of the service instance (`version`), this response includes the OAuth 2 token issuer URI for the service instance (`issuer`), the number of nodes provisioned for the service instance (`nodeCount`), the list of peer service instances, if any (`peers`), and information about the Spring Cloud Services Git repository at build time for the service instance (`git`).

Note: Fields such as those for Git repository information are for diagnostic purposes and intended to provide Pivotal Support with information to help in troubleshooting.

UAA Identity Zones and Clients

Spring Cloud Services uses the multi-tenancy capabilities of the Cloud Foundry User Account and Authentication server (UAA). It creates a new Identity Zone (“the Spring Cloud Services Identity Zone”) for all authorization from Config Server and Service Registry service instances to client applications which have bindings to these service instances.

Within the platform Identity Zone (“the UAA Identity Zone”), Spring Cloud Services creates a `p-spring-cloud-services` UAA client for the SB and a `p-spring-cloud-services-worker` UAA client for the Worker. In the Spring Cloud Services Identity Zone, it creates a `p-spring-cloud-services-worker` UAA client for the Worker.

In the UAA Identity Zone, it also creates the following clients, where `[GUID]` is the service instance’s GUID:

- A `eureka-[GUID]` UAA client per Service Registry service instance.
- A `hystrix-[GUID]` UAA client per Circuit Breaker Dashboard service instance.

In the Spring Cloud Services Identity Zone, it also creates the following clients, where `[GUID]` is the service instance’s GUID:

- A `config-server-[GUID]` UAA client per Config Server service instance.
- A `eureka-[GUID]` UAA client per Service Registry service instance.

Get Access Token for Direct Requests to a Service Instance

To make requests directly against a Config Server service instance’s Spring Cloud Config Server backing application or a Service Registry service instance’s Spring Cloud Netflix Eureka backing application, you must obtain an OAuth 2.0 token. For most Config Server endpoints and for Service Registry endpoints, you can get a client credentials access token using the binding credentials of an application that is bound to the service instance. For the `/encrypt` endpoint on a Config Server service instance, you can get a password credentials access token using the Cloud Foundry Command Line Interface tool (cf CLI).

Get a Client Credentials Access Token

Note: The following procedure uses the jq command-line JSON processing tool.

Run `cf env` , giving the name of an application that is bound to the service instance:


```
$ cf services
Getting services in org myorg / space development as admin...
OK

name           service          plan       bound apps   last operation
config-server  p-config-server  standard   cook         create succeeded

$ cf env cook
Getting env variables for app cook in org myorg / space development as admin...
OK

System-Provided:
{
 "VCAP_SERVICES": {
  "p-config-server": [
   {
    "credentials": {
     "access_token_uri": "https://p-spring-cloud-services.uaa.cf.wise.com/oauth/token",
     "client_id": "p-config-server-876cd13b-1564-4a9a-9d44-c7c8a6257b73",
     "client_secret": "rU7dMUw6bQjR",
     "uri": "https://config-86b38ce0-eed8-4c01-adb4-1a651a6178e2.apps.wise.com"
    },
[...]
```


Then run the following Bash script, which fetches a token and uses the token to access an endpoint on a service instance backing application:

```
TOKEN=$(curl -k ACCESS_TOKEN_URI -u CLIENT_ID:CLIENT_SECRET -d
grant_type=client_credentials | jq -r .access_token); curl -k -H
"Authorization: bearer $TOKEN" -H "Accept: application/json"
URI/ENDPOINT | jq
```

In this script, replace the following placeholders using values from the `cf env` command above:

- `ACCESS_TOKEN_URI` with the value of `credentials.access_token_uri`
- `CLIENT_ID` with the value of `credentials.client_id`
- `CLIENT_SECRET` with the value of `credentials.client_secret`
- `URI` with the value of `credentials.uri`

Replace `ENDPOINT` with the relevant endpoint:

- `application/profile` to retrieve configuration from a Config Server service instance
- `eureka/apps` to retrieve the registry from a Service Registry service instance

Get a Password Credentials Access Token

Run `cf env`, giving the name of an application that is bound to the service instance:

```
$ cf services
Getting services in org myorg / space development as admin...
OK

name            service          plan       bound apps   last operation
config-server   p-config-server  standard   cook         create succeeded

$ cf env cook
Getting env variables for app cook in org myorg / space development as admin...
OK

System-Provided:
{
 "VCAP_SERVICES": {
  "p-config-server": [
   {
    "credentials": {
     "access_token_uri": "https://p-spring-cloud-services.uaa.cf.wise.com/oauth/token",
     "client_id": "p-config-server-876cd13b-1564-4a9a-9d44-c7c8a6257b73",
     "client_secret": "rU7dMUw6bQjR",
     "uri": "https://config-86b38ce0-eed8-4c01-adb4-1a651a6178e2.apps.wise.com"
    },
[...]
```

Copy the value of `credentials.uri`. Then use the `cf oauth-token` command to get the token for the current session. The following example uses curl to access the `/encrypt` endpoint of a Config Server service instance (see the Git Encryption and Encrypted Values section of the Configuring Backends topic):

```
$ curl -H "Authorization: $(cf oauth-token)" https://config-86b38ce0-eed8-4c01-adb4-1a651a6178e2.apps.wise.com/encrypt -d 'Value to be encrypted'
15f826dd703c4a3e9e1d64a5827d3b1f1584a1173c03c42d87e7480dddb07d86e009c2d78a68eee610f7f55c66894907
```

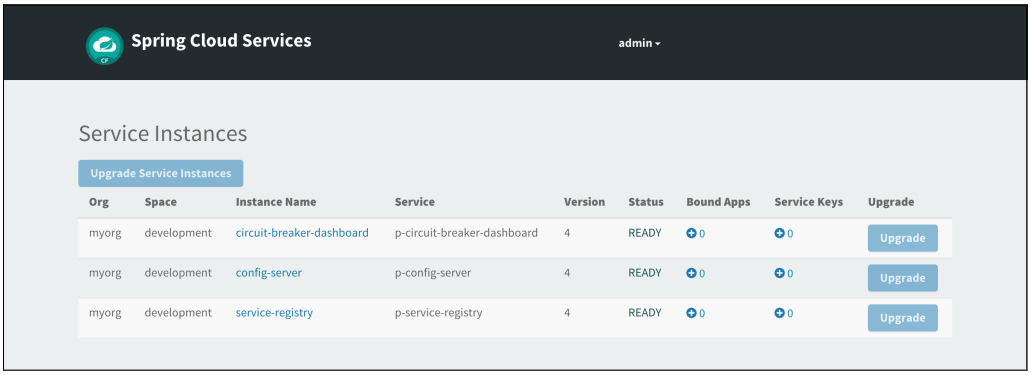
The Service Instances Dashboard

To view the status of individual service instances, visit the Service Instances dashboard. You can access it at the following URL, where `SCS_BROKER_URL` is the URL of the SB (spring-cloud-broker):

```
SCS_BROKER_URL/admin/serviceInstances
```

Locate the SB as described in the Access Broker Applications in Apps Manager section. The Service Instances dashboard is also linked to on the main page of the SB.

The dashboard shows the version and status of each service instance, as well as other information including the number of applications that are bound to the instance. It also provides an **Upgrade** button for each service instance and an **Upgrade Service Instances** button; these buttons may be enabled if any service instances shown in the dashboard are not at the version included with the currently-installed Spring Cloud Services product. For more information about upgrading service instances, see the Service Instance Upgrades topic.



You can click the **Instance Name** of a listed service instance to view the instance's dashboard. Click the **+** icon beside the count of an instance's **Bound Apps** or **Service Keys** to view the names of the applications or service keys that are bound to that instance.

Note: The Service Instances dashboard is updated frequently (close to real-time). If using the cf CLI, you may notice a discrepancy between the status given for a service instance by the cf CLI (e.g., by the `cf service` command) versus that given by the Service Instances dashboard. The status retrieved by the cf CLI is not updated as frequently and may take time to match that shown on the Service Instances dashboard.

Application Health and Status

For more visibility into how Spring Cloud Services service instances and the broker applications are behaving, or for troubleshooting purposes, you may wish to access those applications directly. See below for information about accessing their output.

Read Broker Application Logs

To access logs for the SB and Worker applications, target the "system" org and its "p-spring-cloud-services" space:

```
$ cf target -o system -s p-spring-cloud-services

API endpoint:   https://api.cf.wise.com (API version: 2.43.0)
User:           admin
Org:            system
Space:          p-spring-cloud-services
Ben-Kleins-MacBook-Pro:Work bklein$ cf apps
Getting apps in org system / space p-spring-cloud-services as admin...
OK

name                    requested state   instances   memory   disk   urls
spring-cloud-broker         started           1/1         1G       1G     spring-cloud-broker.apps.wise.com
spring-cloud-broker-worker   started          1/1         1G       1G     spring-cloud-broker-worker.apps.wise.com
```

Then you can use `cf logs` to tail logs for either the SB:

```
$ cf logs spring-cloud-broker
Connected, tailing logs for app spring-cloud-broker in org system / space p-spring-cloud-services as admin...
```

or the Worker:

```
$ cf logs spring-cloud-broker-worker
Connected, tailing logs for app spring-cloud-broker-worker in org system / space p-spring-cloud-services as admin...
```

Read Backing Application Logs

To access logs for service instance backing applications, target the "p-spring-cloud-services" org and its "instances" space:

```
$ cf target -o p-spring-cloud-services -s instances

API endpoint:   https://api.cf.wise.com (API version: 2.43.0)
User:           admin
Org:            p-spring-cloud-services
Space:          instances
$ cf apps
Getting apps in org p-spring-cloud-services / space instances as admin...
OK

name                                    requested state   instances   memory   disk   urls
config-86b38ce0-eed8-4c01-adb4-1a651a6178e2    started          1/1         1G       1G     config-86b38ce0-eed8-4c01-adb4-1a651a6178e2.apps.wise.com
eureka-493b6b17-512a-4961-8f85-6178251fe2fa    started          1/1         1G       1G     eureka-493b6b17-512a-4961-8f85-6178251fe2fa.apps.wise.com
hystrix-ff835bde-5b3a-4623-a57f-8d88028b6376   started          1/1         1G       1G     hystrix-ff835bde-5b3a-4623-a57f-8d88028b6376.apps.wise.com
turbine-ff835bde-5b3a-4623-a57f-8d88028b6376   started          1/1         1G       1G     turbine-ff835bde-5b3a-4623-a57f-8d88028b6376.apps.wise.com
```

Then you can use `cf logs` to tail logs for a backing application.

```
$ cf logs config-86b38ce0-eed8-4c01-adb4-1a651a6178e2
Connected, tailing logs for app config-86b38ce0-eed8-4c01-adb4-1a651a6178e2 in org p-spring-cloud-services / space instances as admin...
```

If the backing application is not running properly then detailed health information will be written to its log on each GET request sent to its `health` Actuator endpoint. No health details are logged if the backing application is healthy.

```
2017-05-04T13:37:17.43+0100 [APP/PROC/WEB/0]OUT 2017-05-04 12:37:17.431  INFO 14 --- [nio-8080-exec-7] i.p.s.c.h.ConfigServerHealthEndpoint     : Health status: DOWN. Health deta
2017-05-04T13:37:17.43+0100 [RTR/0]      OUT config-80593543-d141-424d-bcb9-1aa583be2356.olive.springapps.io - [2017-05-04T12:37:17.067+0000] "GET /health HTTP/1.1" 503 0 390 "
```

Access Actuator Endpoints

The SB and Worker applications, as well as backing applications for service instances, use Spring Boot Actuator. Actuator adds a number of endpoints to these applications; some of the added endpoints are summarized below.

ID	Function
`env`	Displays profiles, properties, and property sources from the application environment
`health`	Displays information about the health and status of the application

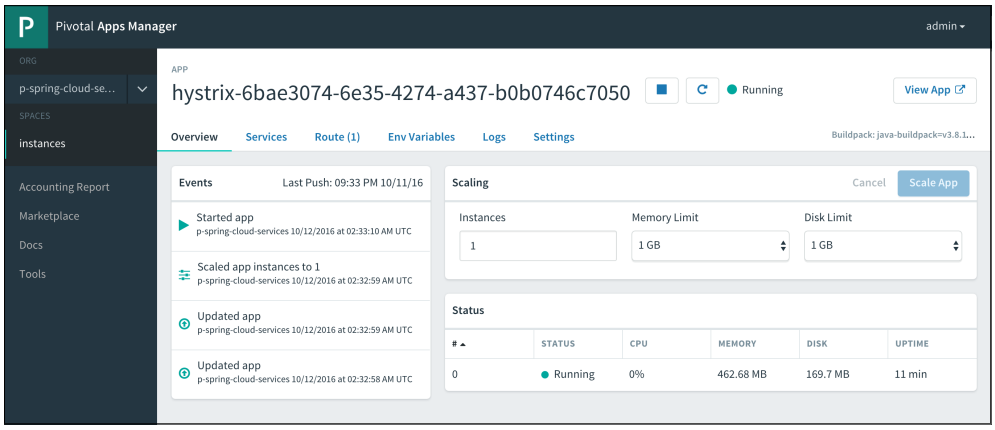
ID	Function
`metrics`	Displays metrics information from the application
`mappings`	Displays list of `@RequestMapping` paths in the application
`shutdown`	Allows for graceful shutdown of the application. Disabled by default; not enabled in Spring Cloud Services broker or instance-backing applications

See the Endpoints section of the Actuator documentation for the full list of endpoints.

Invoking Actuator Endpoints

The SB application, the Worker application, the Service Registry's Eureka backing application, the Config Server's backing application, and the Circuit Breaker Dashboard's Hystrix backing application all use OAuth 2 authentication. To access Actuator endpoints on one of these applications, you must supply a valid OAuth 2 token in the request header.

1. Log in to Apps Manager as an admin user and navigate to the relevant application: for the SB or Worker, access the SB application or Worker application as described in the The Service Broker section; for a Service Registry, Config Server, or Circuit Breaker Dashboard service instance, access the backing application as described in the Service Instance Management section.



2. From the **Route** tab, copy the value of the application's route, as described in the Get Service Instance Information section.

3. In a terminal where you have already authenticated to PCF using `cf auth` or `cf login`, send the request to the Actuator endpoint (e.g., using cURL), appending the endpoint ID to the route URL; e.g., for the `health` endpoint, this would be something like `https://config-86b38ce0-eed8-4c01-adb4-1a651a6178e2.apps.wise.com/health`. The current OAuth 2 token can be obtained using the `cf oauth-token` command. In the example below, the response from `cf oauth-token` is used directly inside the header string.

```
$ curl -k https://config-86b38ce0-eed8-4c01-adb4-1a651a6178e2.apps.wise.com/health -H "Authorization: $(cf oauth-token)"
{"status":"UP","git":{"status":"UP","default":{"status":"UP","repository":{"uri":"https://github.com/pivotal-cf/p-spring-cloud-services-acceptance"}},"app1":{"status":"UP","repository":
```

You also can send a request to each application's `health` Actuator endpoint without the authorization header. In that case, the response JSON object will contain only the summary status:

```
<pre class="terminal">
$ curl -k https://config-86b38ce0-eed8-4c01-adb4-1a651a6178e2.apps.wise.com/health
{"status":"UP"}
</pre>
```

Sending an unauthenticated request to any of the other Actuator endpoints will return a response indicating insufficient authorization.

Invoke Config Server Configuration Endpoints

To view the configuration properties that a Config Server service instance is returning for an application, you can access the configuration endpoints on the service instance's Spring Cloud Config Server backing application directly.

1. Obtain an access token for interacting with the service instance, as described in the Get Access Token for Direct Requests to a Service Instance section. In Step 1 of that section, copy the URL in `credentials.uri` from the service instance's entry in the `VCAP_SERVICES` environment variable.

2. Make a request of the Config Server service instance backing application in the format `http://SERVER/APPLICATION_NAME/PROFILE`, where `SERVER` is the URL from `credentials.uri` as mentioned in Step 1, `APPLICATION_NAME` is the application name as set in the `spring.application.name` property, and `PROFILE` is the name of a profile that has a configuration file in the repository.

   An example request URL might look like this:

   ```
   https://config-86b38ce0-eed8-4c01-adb4-1a651a6178e2.apps.wise.com/cook/production
   ```

   Or, for the `default` profile:

   ```
   https://config-86b38ce0-eed8-4c01-adb4-1a651a6178e2.apps.wise.com/cook/default
   ```

   You can make the request using curl, providing the token—`TOKEN` in the below example—in a header with the `-H` or `--header` option:

```
$ curl -H 'Authorization: bearer TOKEN' https://config-86b38ce0-eed8-4c01-adb4-1a651a6178e2.apps.wise.com/cook/default
{"name":"cook","profiles":["default"],"label":null,"version":"7b5da0d68d9237f2852f7ac6c5e7474fd433c3f3","propertySources":[{"name":"https://github.com/spring-cloud-services-samp
```

For a list of path formats that the Config Server accepts, see the Request Paths section of the Background Information topic in the Config Server documentation.

Service Instance Upgrades

See the below sections for steps to upgrade an individual Spring Cloud Services service instance after an upgrade of the Spring Cloud Services product.

Config Server Upgrades

After an upgrade of the Spring Cloud Services product, follow the below steps to upgrade an individual Config Server service instance.

1. Using the `cf rename-service` command, rename the current Config Server service instance. For example, given a service instance named “config-server”, you might rename it to “config-server-old”.

```
$ cf rename-service config-server config-server-old
Renaming service config-server to config-server-old in org myorg / space development as user...
OK
```

2. Using the `cf create-service` command, create a new instance with the old instance’s settings and former name.

```
$ cf create-service -c '{ "git": { "uri": "https://github.com/spring-cloud-samples/cook-config", "label": "master"  } }' p-config-server standard config-server
Creating service instance config-server in org myorg / space development as user...
OK

Create in progress. Use 'cf services' or 'cf service config-server' to check operation status.
```

3. For each application that has been bound to the service instance, use the `cf rename` command to rename the application. For example, given an application named “cook”, you might rename it to “cook-old”.

```
$ cf rename cook cook-old
Renaming app cook to cook-old in org myorg / space development as user...
OK
```

4. Update client dependencies (see the Client Dependencies topic) in the original application and push it, using a new temporary route, to Pivotal Cloud Foundry (PCF). Ensure that the updated original application is bound to the new service instance.

```
$ cf push -n cook-new cook
Using manifest file manifest.yml

Updating app cook in org myorg / space development as user...
OK

Using route cook-new.wise.com
Uploading cook...
```

5. Verify that the updated original application functions properly, then use the `cf map-route` command to map the original route to the updated original application.

```
$ cf map-route cook wise.com -n cook
Creating route cook.wise.com for org myorg / space development as user...
OK
Route cook.wise.com already exists
Adding route cook.wise.com to app cook in org myorg / space development as user...
OK
```

6. Delete the old application and service instance, then delete the temporary route on the updated application.

```
$ cf delete cook-old

Really delete the app cook-old?> y
Deleting app cook-old in org myorg / space development as user...
OK

$ cf delete-service config-server-old

Really delete the service config-server-old?> y
Deleting service config-server-old in org myorg / space development as user...
OK

$ cf delete-route wise.com -n cook-new

Really delete the route cook-new.wise.com?> y
Deleting route cook-new.wise.com...
OK
```

Service Registry Upgrades

After an upgrade of the Spring Cloud Services product, you can use the `cf update-service` command to update an individual Service Registry service instance.

Run the `cf update-service` command, supplying the `upgrade` and `force` parameters and setting both to `true`:

```
$ cf update-service service-registry -c '{"upgrade": true, "force": true}'
Updating service instance service-registry as user...
OK

Update in progress. Use 'cf services' or 'cf service service-registry' to check operation status.
```

For more information about parameters accepted for an update to a Service Registry service instance, see the Updating an Instance section of the

Managing Service Instances topic in the Service Registry documentation.

Circuit Breaker Dashboard Upgrades

After an upgrade of the Spring Cloud Services product, you can use the `cf update-service` command to update an individual Circuit Breaker Dashboard service instance.

Run the `cf update-service` command, supplying the `upgrade` and `force` parameters and setting both to `true`:

```
$ cf update-service circuit-breaker-dashboard -c '{"upgrade": true, "force": true}'
Updating service instance circuit-breaker-dashboard as user...
OK

Update in progress. Use 'cf services' or 'cf service circuit-breaker-dashboard' to check operation status.
```

For more information about parameters accepted for an update to a Circuit Breaker Dashboard service instance, see the Updating an Instance section of the Managing Service Instances topic in the Circuit Breaker Dashboard documentation.

Client Dependencies

Page last updated:

See below for information about the dependencies required for client applications using Spring Cloud Services service instances.

Include Spring Cloud Services Dependencies

Important: Ensure that the ordering of the Maven BOM dependencies listed below is preserved in your application's build file. Dependency resolution is affected in both Maven and Gradle by the order in which dependencies are declared.

To work with Spring Cloud Services service instances, your client application must include the `spring-cloud-services-dependencies` and `spring-cloud-dependencies` BOMs. Unless you are using the `spring-boot-starter-parent` or Spring Boot Gradle plugin, you must also specify the `spring-boot-dependencies` BOM as a dependency. See below for compatible Spring Boot and Spring Cloud versions to use with a given Spring Cloud Services release.

Spring Cloud Services	Spring Boot	Spring Cloud
1.0.x	1.2.x	Angel.x
1.1.x	1.3.x–1.4.x	Brixton.x
1.2.x	1.3.x–1.4.x	Brixton.x
1.3.x	1.3.x–1.4.x	Camden.x
1.4.x *	1.5.x	Dalston.x

Important: Spring Cloud Services is affected by an issue in Spring Boot version 1.5.3 and is not compatible with that version. To be compatible with Spring Cloud Services 1.4.0, client applications should use Spring Boot versions 1.5.0–1.5.2 or 1.5.4 and later.

Note: An application using Spring Cloud Services 1.3.x dependencies only requires an update to 1.4.x dependencies if using the Config Server support for HashiCorp Vault.

If using Maven, include in `pom.xml`:

```
<parent>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-parent</artifactId>
 <version>1.5.9.RELEASE</version>
 <relativePath/> <!-- lookup parent from repository -->
</parent>

<dependencyManagement>
 <dependencies>
    <dependency>
       <groupId>io.pivotal.spring.cloud</groupId>
       <artifactId>spring-cloud-services-dependencies</artifactId>
       <version>1.5.0.RELEASE</version>
       <type>pom</type>
       <scope>import</scope>
    </dependency>
    <dependency>
       <groupId>org.springframework.cloud</groupId>
       <artifactId>spring-cloud-dependencies</artifactId>
       <version>Dalston.SR5</version>
       <type>pom</type>
       <scope>import</scope>
    </dependency>
 </dependencies>
</dependencyManagement>
```

If not using the `spring-boot-starter-parent`, include in the `<dependencyManagement>` block of `pom.xml`:

```
<dependencyManagement>
 <dependencies>
    <dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-dependencies</artifactId>
       <version>1.5.9.RELEASE</version>
       <type>pom</type>
       <scope>import</scope>
    </dependency>

    <!-- ... -->

 </dependencies>
</dependencyManagement>
```

If using Gradle, you will also need to use the Gradle dependency management plugin.

Include in `build.gradle`:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("io.spring.gradle:dependency-management-plugin:1.0.2.RELEASE")
        classpath("org.springframework.boot:spring-boot-gradle-plugin:1.5.9.RELEASE")
    }
}

apply plugin: "java"
apply plugin: "org.springframework.boot"
apply plugin: "io.spring.dependency-management"

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:Dalston.SR5"
        mavenBom "io.pivotal.spring.cloud:spring-cloud-services-dependencies:1.5.0.RELEASE"
    }
}

repositories {
    maven {
        url "https://repo.spring.io/plugins-release"
    }
}
```

If not using the Spring Boot Gradle plugin, include in the `dependencyManagement` block of `build.gradle`:

```
dependencyManagement {
    imports {
        mavenBom "org.springframework.boot:spring-boot-dependencies:1.5.9.RELEASE"
    }
}
``
```

Config Server

Your application must declare `spring-cloud-services-starter-config-client` as a dependency.

If using Maven, include in `pom.xml`:

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

If using Gradle, include in `build.gradle`:

```
dependencies {
    compile("io.pivotal.spring.cloud:spring-cloud-services-starter-config-client")
}
```

Service Registry

Your application must declare `spring-cloud-services-starter-service-registry` as a dependency.

If using Maven, include in ` `pom.xml`:

```
<dependencies>
  <dependency>
    <groupId>io.pivotal.spring.cloud</groupId>
    <artifactId>spring-cloud-services-starter-service-registry</artifactId>
  </dependency>
</dependencies>
```

If using Gradle, include in `build.gradle`:

```
dependencies {
    compile("io.pivotal.spring.cloud:spring-cloud-services-starter-service-registry")
}
```

Circuit Breaker Dashboard

Your application must declare `spring-cloud-services-starter-circuit-breaker` as a dependency.

If using Maven, include in `pom.xml`:

```
<dependencies>
  <dependency>
    <groupId>io.pivotal.spring.cloud</groupId>
    <artifactId>spring-cloud-services-starter-circuit-breaker</artifactId>
  </dependency>
</dependencies>
```

If using Gradle, include in `build.gradle`:

```
dependencies {
    compile("io.pivotal.spring.cloud:spring-cloud-services-starter-circuit-breaker")
}
```

Dependency Versioning

A release of the Spring Cloud Services client dependencies follows a release of the underlying open-source Spring Cloud components, which are based on particular releases of Spring Boot. For more information about the basis of Spring Cloud or Spring Cloud Services library releases, see below.

Spring Cloud OSS Releases

Each Spring Cloud major release is named after a London Underground station (as in “Spring Cloud Camden.RELEASE”). Minor releases are known as service releases and are designated `SR[n]` (as in “Spring Cloud Camden.SR7”), where `[n]` is the minor release number.

A given major release of Spring Cloud supports the current Spring Boot minor release and one Spring Boot minor release following. When a new Spring Boot major release is published, it will be supported only in the next major release of Spring Cloud. (The Spring Boot 2.0 release follows the Spring Boot 1.5 release; there was no 1.6 release.)

Spring Cloud Release	Supported Spring Boot Release
Camden	1.4–1.5
Dalston	1.5
Edgware	1.5
Finchley	2.0

Spring Cloud Services Client Releases

The Spring Cloud Services connectors, as well as the starters dependencies that include the connectors, are versioned and released independently of the Spring Cloud Services tile. The connectors and starters are based on the open-source Spring Cloud client libraries for Config Server, Netflix Eureka, and Netflix Hystrix, and have a new release to follow each Spring Cloud release that affects these components.

A given minor release of the Spring Cloud Services starters uses a particular release of Spring Cloud.

Spring Cloud Services Starters Release	Underlying Spring Cloud Release
1.2.0	Brixton.SR6
1.3.0	Camden.SR2
1.4.0	Camden.SR3
1.5.0	Dalston.SR1

Spring Cloud Services Tile Releases

The Spring Cloud Services tile is versioned and released independently of the Spring Cloud Services connectors and starters. The tile contains the server-side components from Spring Cloud for Config Server, Netflix Eureka, Netflix Hystrix, and Netflix Hystrix Dashboard, and typically has a new release following each Spring Cloud service release that affects these components.

A given minor release of the tile includes server-side components from a particular release of Spring Cloud, typically the last stable release.

Spring Cloud Services Tile Release	Underlying Spring Cloud Release
1.1.x	Brixton
1.2.x	Brixton
1.3.x	Camden
1.4.x	Dalston

Troubleshooting Client Applications

See below for information about troubleshooting problems in client applications which are bound to Spring Cloud Services service instances.

“Can’t contact any eureka nodes - possibly a security group issue?”

A client application bound to a Service Registry service instance may log an error such as the following:

```
2015-10-01T11:33:38.43-0500 [App/0]      OUT 2015-10-01 16:33:38.433  WARN 29 ---
[           main] com.netflix.discovery.DiscoveryClient    : Action: Refresh  =>
returned status of 401 from https://eureka-aa9d8079f0f3.example.com/eureka/apps/
2015-10-01T11:33:38.43-0500 [App/0]      OUT 2015-10-01 16:33:38.438 ERROR 29 ---
[           main] com.netflix.discovery.DiscoveryClient    : Can't get a response from
https://eureka-aa9d8079f0f3.example.com/eureka/apps/
2015-10-01T11:33:38.43-0500 [App/0]      OUT Can't contact any eureka nodes - possibly a
security group issue?
2015-10-01T11:33:38.43-0500 [App/0]      OUT java.lang.RuntimeException: Bad status: 401
2015-10-01T11:33:38.43-0500 [App/0]      OUT    at
com.netflix.discovery.DiscoveryClient.makeRemoteCall(DiscoveryClient.java:1155)
```

If your PCF environment is using a self-signed certificate (such as a certificate generated in Elastic Runtime), you must set the `TRUST_CERTS` environment variable on your application to the API endpoint of Elastic Runtime. See the Add Self-Signed SSL Certificate to JVM Truststore section of the Writing Client Applications topic in the Service Registry documentation.

“sun.security.validator.ValidatorException: PKIX path building failed”

If you encounter the following exception:

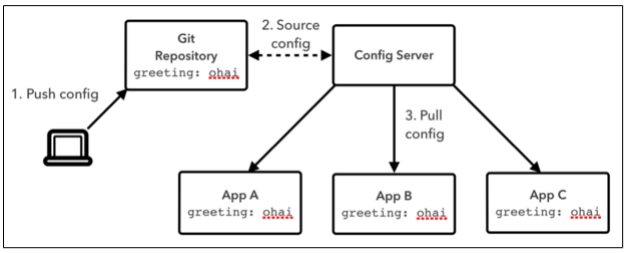
```
sun.security.validator.ValidatorException: PKIX path building failed:
sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid
certification path to requested target
```

If your PCF environment is using self-signed certificates, make sure that you have set the `TRUST_CERTS` environment variable on the application to the Elastic Runtime API endpoint as described in the Add Self-Signed SSL Certificate to JVM Truststore section of the Writing Client Applications topic in the Service Registry documentation.

Config Server for Pivotal Cloud Foundry

Overview

Config Server for Pivotal Cloud Foundry (PCF) is an externalized application configuration service, which gives you a central place to manage an application's external properties across all environments. As an application moves through the deployment pipeline from development to test and into production, you can use Config Server to manage the configuration between environments and be certain that the application has everything it needs to run when you migrate it. Config Server easily supports labelled versions of environment-specific configurations and is accessible to a wide range of tooling for managing the content.



The concepts on both client and server map identically to the Spring `Environment` and `PropertySource` abstractions. They work very well with Spring applications, but can be applied to applications written in any language. The default implementation of the server storage backend uses Git. HashiCorp Vault is also supported.

Config Server for Pivotal Cloud Foundry is based on Spring Cloud Config Server. For more information about Spring Cloud Config and about Spring configuration, see Additional Resources.

Refer to the "Cook" sample application to follow along with code in this section.

Background Information

Page last updated:

The Config Server

The Config Server serves configurations stored as either Java Properties files or YAML files. It reads files from a Git repository (a *configuration source*). Given the URI of a configuration source, the server will clone the repository and make its configurations available to client applications in JSON as a series of `propertySources`.

Configuration Sources

A configuration source contains one or more configuration files used by one or more applications. Each file applies to an *application* and can optionally apply to a specific *profile* and / or *label*.

The following is the structure of a Git repository which could be used as a configuration source.

```
master
------
https://github.com/myorg/configurations
|- myapp.yml
|- myapp-development.yml
|- myapp-production.yml

tag v1.0.0
----------
https://github.com/myorg/configurations
|- myapp.yml
|- myapp-development.yml
|- myapp-production.yml
```

In this example, the configuration source defines configurations for the `myapp` application. The Server will serve different properties for `myapp` depending on the values of `{profile}` and `{label}` in the request path. If the `{profile}` is neither `development` nor `production`, the server will return the properties in `myapp.yml`, or if the `{profile}` is `production`, the server will return the properties in both `myapp-production.yml` and `myapp.yml`.

`{label}` can be a Git commit hash as well as a tag or branch name. If the request contains a `{label}` of (e.g.) `v1.0.0`, the Server will serve properties from the `v1.0.0` tag. If the request does not contain a `{label}`, the Server will serve properties from the default label. For Git repositories, the default label is **master**. You can reconfigure the default label (see the Configuring with Git topic).

Request Paths

Configuration requests use one of the following path formats:

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

A path includes an *application*, a *profile*, and optionally a *label*.

- **Application**: The name of the application. In a Spring application, this will be derived from `spring.application.name` or `spring.cloud.config.name`.
- **Profile**: The name of a profile, or a comma-separated list of profile names. The Config Server's concept of a "profile" corresponds directly to that of the Spring Profile.
- **Label**: The name of a version marker in the configuration source (the repository). This might be a branch name, a tag name, or a Git commit hash. The value given to the Config Server as a default label (the setting of `git.label`; see the table of the Config Server's general configuration parameters) can be overridden in a client application by setting the `spring.cloud.config.label` property.

For information about using a Cloud Foundry application as a Config Server client, see the Configuration Clients section.

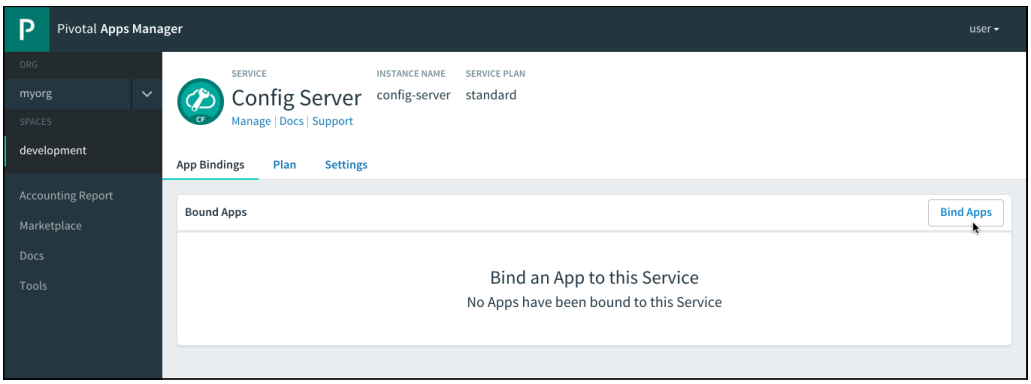
Configuration Clients

Config Server client applications can be written in any language. The interface for retrieving configuration is HTTP, and the endpoints are protected by OAuth 2.0.

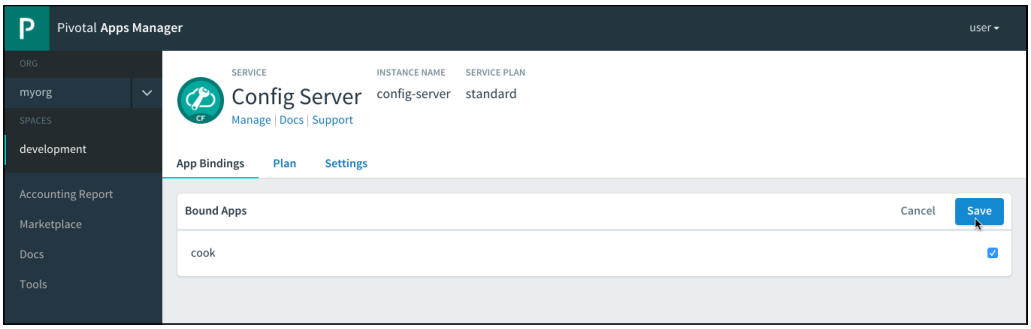
To be given a base URI and client credentials for accessing a Config Server instance, a Cloud Foundry application needs to bind to the instance.

Bind an Application to a Service Instance

Click the service instance's listing in Apps Manager. Under **App Bindings**, click **Bind Apps**.



Select an application and click **Save**.



Alternatively, you can use the Cloud Foundry Command Line Interface tool (cf CLI). Run the command `cf bind-service`, specifying the application name and service instance name.

```
$ cf bind-service cook config-server
Binding service config-server to app cook in org myorg / space development as admin...
OK
TIP: Use 'cf restage cook' to ensure your env variable changes take effect
```

Then, as the command output suggests, run the command `cf restage` to restage the application before proceeding. (See the next section for information about the environment variable to which the command output refers.)

```
$ cf restage cook
Restaging app cook in org myorg / space development as admin...
...
```

Configuration Requests and Responses

After an application is bound to the service instance, the application's `VCAP_SERVICES` environment variable will contain an entry under the key `p-config-server`. You can view the application's environment variables using the cf CLI:

```
$ cf env cook
Getting env variables for app cook in org myorg / space development as admin...
OK

System-Provided:
{
 "VCAP_SERVICES": {
  "p-config-server": [
   {
    "credentials": {
     "access_token_uri": "https://p-spring-cloud-services.uaa.cf.wise.com/oauth/token",
     "client_id": "p-config-server-c4a56a3d-9507-4c2f-9cd1-f858dbf9e11c",
     "client_secret": "9aGx9K5Vx0cM",
     "uri": "https://config-51711835-4626-4823-b5a1-e5d91012f3f2.apps.wise.com"
    },
    "label": "p-config-server",
    "name": "config-server",
    "plan": "standard",
    "tags": [
     "configuration",
     "spring-cloud"
    ]
   }
  ]
 }
}
```

The following is an example of a response from the Config Server to a request using the path `/cook/production` (where the application is `cook` and the profile is `production`).

```
{
  "name":"cook",
  "profiles":[
    "production"
  ],
  "label":null,
  "version":"7b5da0d68d9237f2852f7ac6c5e7474fd433c3f3",
  "propertySources":[
    {
      "name":"https://github.com/spring-cloud-services-samples/cook-config/cook-production.properties",
      "source":
        {
          "cook.special":"Cake a la mode"
        }
    },
    {
      "name":"https://github.com/spring-cloud-services-samples/cook-config/cook.properties",
      "source":
        {
          "cook.special":"Pickled Cactus"
        }
    }
  ]
}
```


As shown in the above example, the Config Server may include multiple values for the same property in its response. In that case, the client application must decide how to interpret the response; the intent is that the first value in the list should take precedence over the others. Spring applications will do this for you automatically.

Spring Client Applications

A Spring application can use a Config Server as a property source. Properties from a Config Server will override those defined locally (e.g. via an `application.yml` in the classpath).

The application requests properties from the Config Server using a path such as `/{application}/{profile}/[{label}]` (see the Request Paths section). It will derive values for these three parameters from the following properties:

- `{application}`: `spring.cloud.config.name` or `spring.application.name`.
- `{profile}`: `spring.cloud.config.env` or `spring.profiles.active`.
- `{label}`: `spring.cloud.config.label` if it is defined; otherwise, the Config Server's default label.

These values can be specified in an `application.yml` or `application.properties` file on the classpath, via a system property (as in `-Dspring.profiles.active=production`), or (more commonly in Cloud Foundry) via an environment variable:

```
$ cf set-env cook SPRING_PROFILES_ACTIVE production
```

Given the above example response for the request path `/cook/production`, a Spring application would give the two Config Server property sources precedence over other property sources. This means that properties from `https://github.com/myorg/configurations/cook-production.yml` would have precedence over properties from `https://github.com/myorg/configurations/cook.yml`, which would have precedence over properties from the application's other property sources (such as `classpath:application.yml`).

For a specific example of using a Spring application as a Config Server client, see the Writing Client Applications topic.

Managing Service Instances

Page last updated:

See below for information about managing Config Server service instances.

Creating an Instance

You can create a Config Server service instance using either the Cloud Foundry Command Line Interface tool (cf CLI) or Pivotal Cloud Foundry® Apps Manager (you cannot configure a service instance using Apps Manager).

Using the cf CLI

Begin by targeting the correct org and space.

```
$ cf target -o myorg -s development

API endpoint:   https://api.cf.wise.com (API version: 2.43.0)
User:           user
Org:            myorg
Space:          development
```

You can view plan details for the Config Server product using `cf marketplace -s`.

```
$ cf marketplace
Getting services from marketplace in org myorg / space development as user...
OK

service                     plans          description
p-circuit-breaker-dashboard   standard       Circuit Breaker Dashboard for Spring Cloud Applications
p-config-server               standard       Config Server for Spring Cloud Applications
p-mysql                     100mb-dev      MySQL service for application development and testing
p-rabbitmq                  standard       RabbitMQ is a robust and scalable high-performance multi-protocol messaging broker.
p-service-registry            standard       Service Registry for Spring Cloud Applications

TIP:  Use 'cf marketplace -s SERVICE' to view descriptions of individual plans of a given service.

$ cf marketplace -s p-config-server
Getting service plan information for service p-config-server as user...
OK

service plan   description     free or paid
standard       Standard Plan   free
```

Create the service instance using `cf create-service`, using the `-c` flag to provide a JSON object that specifies the configuration parameters. For information about the parameters used to configure a Git configuration source, see the Configuring with Git topic. For information about the parameters used to configure a HashiCorp Vault configuration source, see the Configuring with Vault topic.

General parameters accepted for the Config Server are listed below.

Parameter	Function	Example
`count`	The number of nodes to provision: 1 by default, more for running in high-availability mode	`'{"count": 3}'`

To create an instance, specifying settings for Git configuration sources and that three nodes should be provisioned:

```
$ cf create-service -c '{"git": { "uri": "https://github.com/spring-cloud-samples/config-repo", "repos": { "cook": { "pattern": "cook*", "uri": "https://github.com/spring-cloud-services-samples/coo
Creating service instance config-server in org myorg / space development as user...
OK

Create in progress. Use 'cf services' or 'cf service config-server' to check operation status.
```

As the command output suggests, you can use the `cf services` or `cf service` commands to check the status of the service instance. When the service instance is ready, the `cf service` command will give a status of `create succeeded`:

```
$ cf service config-server

Service instance: config-server
Service: p-config-server
Bound apps:
Tags:
Plan: standard
Description: Config Server for Spring Cloud Applications
Documentation url: http://docs.pivotal.io/spring-cloud-services/
Dashboard: https://spring-cloud-broker.apps.wise.com/dashboard/p-config-server/9281c950-9b07-495e-9af4-f9fbd277037d

Last Operation
Status: create succeeded
Message:
Started: 2016-06-24T21:57:21Z
Updated: 2016-06-24T21:59:24Z
```

Important: The `cf service` and `cf services` commands may report a `create succeeded` status even if the Config Server cannot initialize using the

provided settings. For example, given an invalid URI for a configuration source, the service instance may still be created and have a `create succeeded` status.

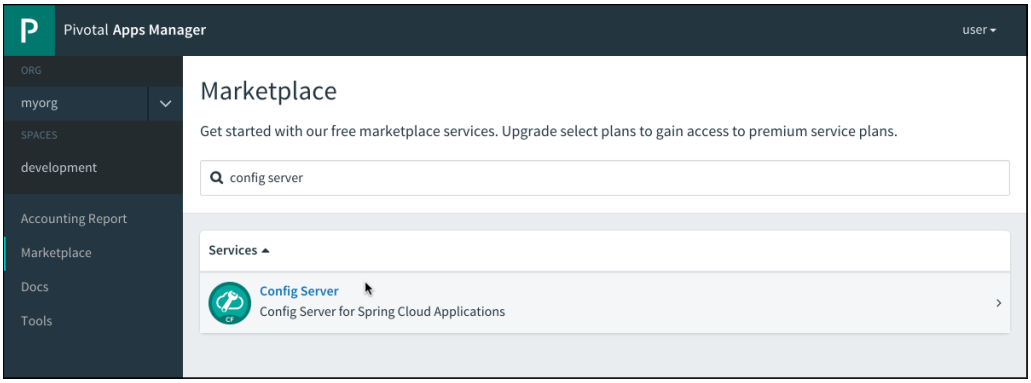
If the service instance does not appear to be functioning correctly, you can visit its dashboard to double-check that the provided settings are valid and accurate. See the Using the Dashboard topic.

Note: You may notice a discrepancy between the status given for a service instance by the cf CLI (e.g., by the `cf service` command) versus that shown on the Service Instances dashboard. The dashboard is updated frequently (close to real-time); the status retrieved by the cf CLI is not updated as frequently and may take time to match the dashboard.

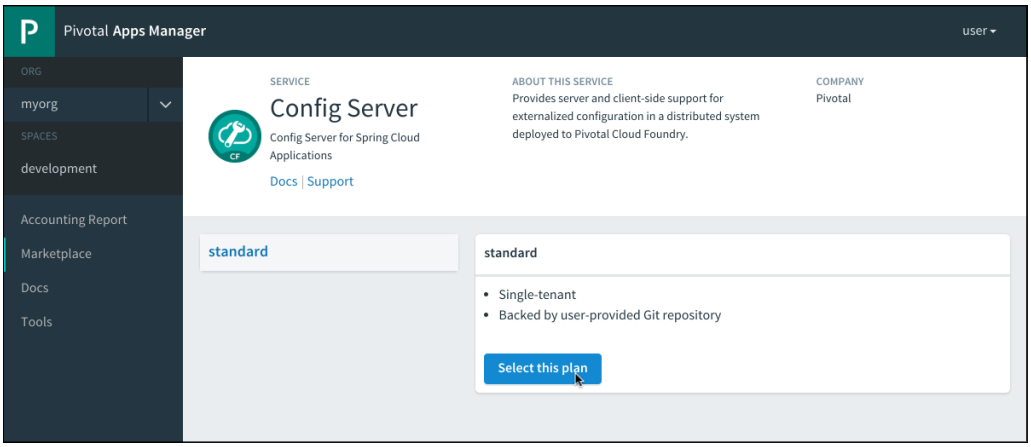
Using Apps Manager

Note: If you create a Config Server service instance using Apps Manager, you will need to use the cf CLI to update the service instance and provide settings (including one or more configuration sources) before the instance will be usable. For information on updating a Config Server service instance, see the Updating an Instance section.

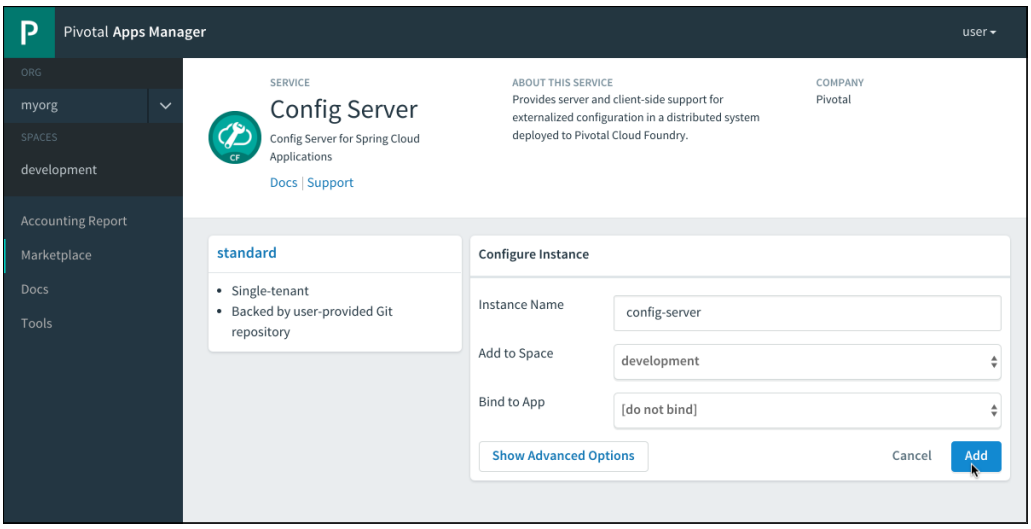
Log into Apps Manager as a Space Developer. In the Marketplace, select **Config Server**.



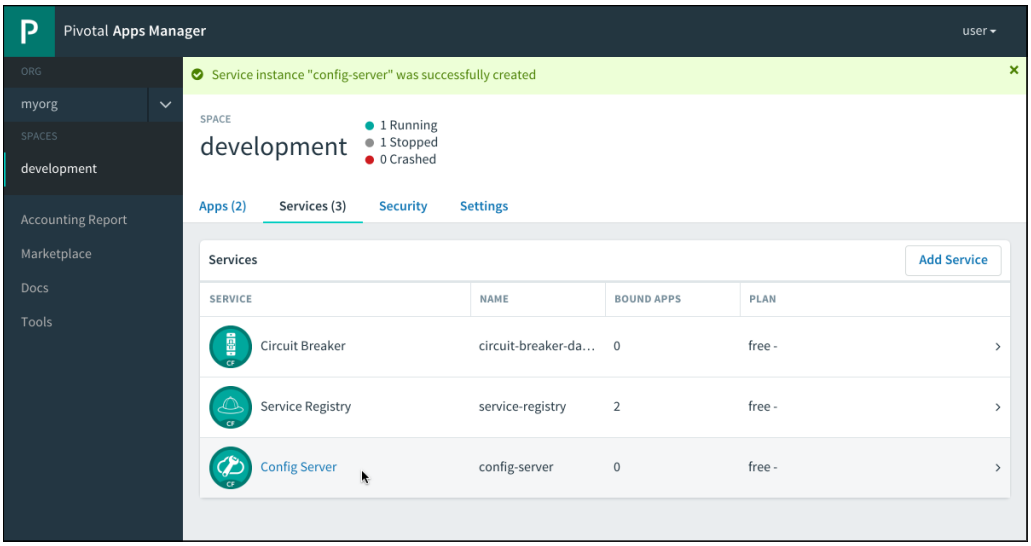
Select the desired plan for the new service instance.



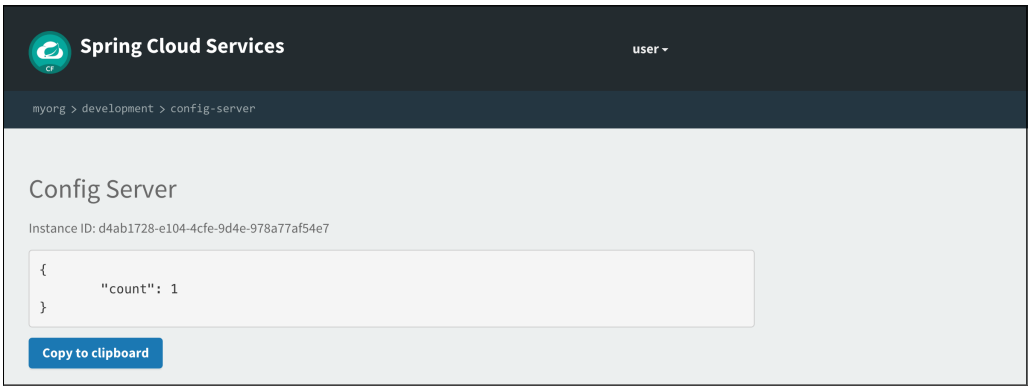
Provide a name for the service instance (for example, “config-server”). Click the **Add** button.



In the **Services** list, click the **Manage** link under the listing for the new service instance.



It may take a few minutes to provision the service instance; while it is being provisioned, you will see a “The service instance is initializing” message. When the instance is ready, its dashboard will load automatically.



The dashboard displays the current settings for the Config Server service instance. Before you can use this service instance, you must update it using the cf CLI to supply it with a configuration source. For information about updating a service instance to configure instance settings, see the Updating an Instance section.

Updating an Instance

You can update settings on a Config Server service instance using the Cloud Foundry Command Line Interface tool (cf CLI). The `cf update-service` command can be given a `-c` flag with a JSON object containing parameters used to configure the service instance.

To update a Config Server service instance’s settings, target the org and space of the service instance:

```
$ cf target -o myorg -s development

API endpoint:   https://api.cf.wise.com (API version: 2.43.0)
User:           user
Org:            myorg
Space:          development
```

Then run `cf update-service SERVICE_NAME -c '{ "PARAMETER": "VALUE" }'`, where `SERVICE_NAME` is the name of the service instance, `PARAMETER` is a supported parameter, and `VALUE` is the value for the parameter. For information about the parameters used to configure a Git configuration source, see the Configuring with Git topic. For information about the parameters used to configure a HashiCorp Vault configuration source, see the Configuring with Vault topic.

General parameters accepted for the Config Server are listed below.

Parameter	Function	Example
`count`	The number of nodes to provision: 1 by default, more for running in high-availability mode	`'{"count": 3}'`
`upgrade`	Whether to upgrade the instance	`'{"upgrade": true}'`
`force`	When `upgrade` is set to `true`, whether to force an upgrade of the instance, even if the instance is already at the latest available service version	`'{"force": true}'`

To update a service instance, setting Git configuration sources, run:

```
$ cf update-service config-server -c '{"git": { "uri": "https://github.com/spring-cloud-samples/config-repo", "repos": { "cook": { "pattern": "cook*", "uri": "https://github.com/spring-cloud-service
Updating service instance config-server as admin...
OK

Update in progress. Use 'cf services' or 'cf service config-server' to check operation status.
```

To update a service instance and set the count of instances for running in high-availability mode, run:

```
$ cf update-service config-server -c '{"count": 3}'
Updating service instance config-server as admin...
OK

Update in progress. Use 'cf services' or 'cf service config-server' to check operation status.
```

As the command output suggests, you can use the `cf services` or `cf service` commands to check the status of the service instance. When the update is complete, the `cf service` command will give a status of `update succeeded`:

```
$ cf service config-server

Service instance: config-server
Service: p-config-server
Bound apps:
Tags:
Plan: standard
Description: Config Server for Spring Cloud Applications
Documentation url: http://docs.pivotal.io/spring-cloud-services/
Dashboard: https://spring-cloud-broker.apps.wise.com/dashboard/p-e2615a44-66f5-4fda-9ddc-079af1db9ae4

Last Operation
Status: update succeeded
Message:
Started: 2016-06-24T21:11:53Z
Updated: 2016-06-24T21:13:59Z
```

Important: The `cf service` and `cf services` commands may report an `update succeeded` status even if the Config Server cannot initialize using the provided settings. For example, given an invalid URI for a configuration source, the service instance may still be restarted and have an `update succeeded` status.

If the service instance does not appear to be functioning correctly after an update, you can visit its dashboard to double-check that the provided settings are valid and accurate. See the Using the Dashboard topic.

The service instance is now updated and ready to be used. For information about using an application to access configuration values served by a Config Server service instance, see the Writing Client Applications topic.

Configuring with Git

Page last updated:

Overview

Git is a distributed version control system (DVCS). It encourages parallel development through simplified branching and merging, optimizes performance by conducting many operations on the local copy of the repository, and uses SHA-1 hashes for checksums to assure integrity and guard against corruption of repository data. For more information about Git, see the documentation.

Spring Cloud Config provides a Git backend so that the Spring Cloud Config Server can serve configuration stored in Git. The Spring Cloud Services Config Server supports this backend and can serve configuration stored in Git to client applications when given the URL to a Git repository (for example, the URL of a repository hosted on GitHub or Bitbucket). For more information about Spring Cloud Config's Git backend, see the documentation.

See below for information about configuring a Config Server service instance to use Git for configuration sources.

General Configuration

Parameters used to configure configuration sources are part of a JSON object called `git`, as in `{"git": { "uri": "http://example.com/config" } }`. For more information on the purposes of these fields, see the The Config Server section of the Background Information topic.

General parameters used to configure the Config Server's default configuration source are listed below.

Parameter	Function
`uri`	The URI (`http://`, `https://`, or `ssh://`) of a repository that can be used as the default configuration source
`label`	The default "label" that can be used with the default repository if a request is received without a label (e.g., if the `spring.cloud.config.label` property is not set in a client application)
`cloneOnStart`	Whether the Config Server should clone the default repository when it starts up (by default, the Config Server will only clone the repository when configuration is first requested from the repository). Valid values are `true` and `false`
`username`	The username used to access the default repository (if protected by HTTP Basic authentication)
`password`	The password used to access the default repository (if protected by HTTP Basic authentication)
`skipSslValidation`	For a `https://` URI, whether to skip validation of the SSL certificate on the default repository's server. Valid values are `true` and `false`

Important: If you set `cloneOnStart` to `true` for a service instance that uses a repository which is secured with HTTP Basic authentication, you must set the `username` and `password` at the same time as you set `cloneOnStart`. Otherwise, the Config Server will be unable to access the repository and the service instance may fail to initialize.

The `uri` setting is required; you cannot define a Config Server configuration source without including a `uri`.

The default value of the `label` setting is `master`. You can set `label` to a branch name, a tag name, or a specific Git commit hash.

To set `label` to point to the `develop` branch of a repository, you might configure settings as shown in the following JSON:

```
'{"git": { "uri": "https://github.com/myorg/config-repo", "label": "develop" } }'
```

To set `label` to point to the `v1.1` tag in a repository, you might configure settings as shown in the following JSON:

```
'{"git": { "uri": "https://github.com/myorg/config-repo", "label": "v1.1" } }'
```

Within a client application, you can override the Config Server's `label` setting by setting the `spring.cloud.config.label` property (for example, in `bootstrap.yml`).

```
spring:
  cloud:
    config:
      label: v1.2
```

Passwords are masked in the Config Server dashboard.

Encryption and Encrypted Values

The Config Server can serve encrypted property values from a configuration file. If the Config Server is configured with a symmetric or asymmetric encryption key and the encrypted values are prefixed with the string `{cipher}`, the Config Server will decrypt the values before serving them to client applications. The Config Server has an `/encrypt` endpoint, which can be used to encrypt property values.

To use these features in a client application, you must use a Java buildpack which contains the Java Cryptography Extension (JCE) Unlimited Strength policy files. These files are contained in the Cloud Foundry Java buildpack from version 3.7.1.

If you cannot use version 3.7.1 or later, you can add the JCE Unlimited Strength policy files to an earlier version of the Cloud Foundry Java buildpack. Fork the buildpack on GitHub, then download the policy files from Oracle and place them in the buildpack's `resources/open_jdk_jre/lib/security` directory. Follow the instructions in the Managing Custom Buildpacks topic to add this buildpack to Pivotal Cloud Foundry. Be sure that it has the lowest position of all enabled Java buildpacks.

If you wish to use public-key (or asymmetric) encryption, you must configure the Config Server to use a PEM-encoded keypair. You might generate such a keypair using, for example, OpenSSL on the command line:

```
$ openssl genpkey -algorithm RSA -outform PEM -pkeyopt rsa_keygen_bits:2048
```

When the Config Server has been configured to encrypt values, you can make a POST request to the `/encrypt` endpoint. Include the property value in the request. A request to encrypt a value might look something like the following (using curl), where `TOKEN_STRING` is an OAuth 2.0 token and `SERVER` is the URL of the Config Server:

```
curl -H 'Authorization: bearer TOKEN_STRING' http://SERVER/encrypt -d 'Value to be encrypted'
```

The Config Server returns the encrypted value. You can use the encrypted value in a configuration file as described in the Encrypted Configuration section of the Configuration Properties topic.

The parameters used to configure server-side encryption for a Config Server are listed below.

Parameter	Function
`encrypt.key`	The key to use for encryption.

To configure a Config Server service instance that can encrypt property values, use the following JSON object:

```
'{"git": {"uri": "https://github.com/spring-cloud-services-samples/cook-config.git" }, "encrypt": { "key": "KEY" }}'
```

If you wish to use public-key (or asymmetric) encryption, you may configure the Config Server to use a PEM-encoded keypair. You might generate such a keypair using, for example, OpenSSL on the command line:

```
$ openssl genpkey -algorithm RSA -outform PEM -pkeyopt rsa_keygen_bits:2048 > server.key
# Translate PEM encoded private key to a RSA private key and remove newlines from it
$ openssl rsa -in server.key | tr -d '\n' > server_rsa.key
```

To configure a Config Server service instance that can encrypt property values with an assymetric keypair, use the following JSON object(key from content of server_rsa.key file):

```
'{"git": {"uri": "https://github.com/spring-cloud-services-samples/cook-config.git" }, "encrypt": { "key": "-----BEGIN RSA PRIVATE KEY-----MIIE.....-----END RSA PRIVATE KEY-----"
}}'
```

The encryption key is masked in the Config Server dashboard.

SSH Repository Access

You can configure a Config Server configuration source so that the Config Server accesses it using the Secure Shell (SSH) protocol. To do so, you must specify a URI using the `ssh://` URI scheme or the Secure Copy Protocol (SCP) style URI format, and you must supply a private key. You may also supply a host key with which the server will be identified. If you do not provide a host key, the Config Server will not verify the host key of the configuration source's server.

A SSH URI must include a username, host, and repository path. This might be specified as shown in the following JSON:

```
'{"git": { "uri": "ssh://git@github.com/spring-cloud-services-samples/cook.git"} }'
```

An equivalent SCP-style URI might be specified as shown in the following JSON:

```
'{"git": { "uri": "git@github.com:spring-cloud-services-samples/cook-config.git"} }'
```

The parameters used to configure SSH for a Config Server configuration source's URI are listed below.

Parameter	Function
`hostKey`	The host key of the Git server. If you have connected to the server via git on the command line, this is in your `.ssh/known_hosts`. Do not include the algorithm prefix; this is specified in `hostKeyAlgorithm`. (Optional.)
`hostKeyAlgorithm`	The algorithm of `hostKey`: one of "ssh-dss", "ssh-rsa", "ecdsa-sha2-nistp256", "ecdsa-sha2-nistp384", and "ecdsa-sha2-nistp521". (Required if supplying `hostKey`.)
`privateKey`	The private key that identifies the Git user, with all newline characters replaced by `\n`. Passphrase-encrypted private keys are not supported.
`strictHostKeyChecking`	Whether the Config Server should fail to start if it encounters an error when using the provided `hostKey`. (Optional.) Valid values are `true` and `false`

To configure a Config Server service instance that uses SSH to access a configuration source, allowing for host key verification, use the following JSON object:

```
'{"git": { "uri": "ssh://git@github.com/spring-cloud-services-samples/cook.git", "hostKey": "AAAAB3NzaC1yc2EAAAABIwAAAQEAq2A7hRGmdnm9tUDbO9IDSwBK6TbQa+...", "hostKeyAlgorithm": "ssh-rsa", "privateKey": "-----BEGIN EXAMPLE RSA PRIVATI
```

To configure a Config Server service instance that uses SSH to access a configuration source, without host key verification, use the following JSON object:

```
'{"git": { "uri": "ssh://git@github.com/spring-cloud-services-samples/cook.git", "privateKey": "-----BEGIN EXAMPLE RSA PRIVATE KEY-----\nMIIJKQIB..."} }'
```

Host and private keys are masked in the Config Server dashboard.

Multiple Repositories

You can configure a Config Server service instance to use multiple configuration sources, which will be used only for specific applications or for applications which are using specific profiles. To do so, you must provide parameters in repository objects within the `git.repos` JSON object. Most parameters set in the `git` object for the default configuration source are also available for specific configuration sources and can be set in repository objects within the `git.repos` object.

Each repository object in the `git.repos` object has a name. In the repository specified in the following JSON, the name is “cookie”:

```
'{ "git": { "repos": { "cookie": { "uri": "https://github.com/spring-cloud-services-samples/cook-config" } } } }'
```

Each repository object also has a `pattern`, which is a comma-separated list of application and profile names separated with forward slashes (`/`, as in `app/profile`) and potentially including wildcards (`*`, as in `app*/profile*`). If you do not supply a pattern, the repository object’s name will be used as the pattern. In the repository specified in the following JSON, the pattern is `co*/dev*` (matching any application whose name begins with `co` and which is using a profile whose name begins with `dev`), and the default pattern would be `cookie`:

```
'{"git": { "repos": { "cookie": { "pattern": "co*/dev*", "uri": "https://github.com/spring-cloud-services-samples/cook-config" } } } }'
```

For more information about the pattern format, see “Pattern Matching and Multiple Repositories” in the Spring Cloud Config documentation.

The parameters used to configure specific configuration sources for the Config Server are listed below.

Parameter	Function
`repos."name"`	A repository object, containing repository fields
`repos."name".pattern`	A pattern for the names of applications that store configuration from this repository (if not supplied, will be `"name"`)
`repos."name".uri`	The URI (`http://`, `https://`, or `ssh://`) of this repository
`repos."name".label`	The default “label” to use with this repository if a request is received without a label (e.g., if the `spring.cloud.config.label` property is not set in a client application)
`repos."name".searchPaths`	A pattern used to search for configuration-containing subdirectories in this repository
`repos."name".cloneOnStart`	Whether the Config Server should clone this repository when it starts up (by default, the Config Server will only clone the repository when configuration is first requested from the repository). Valid values are `true` and `false`
`repos."name".username`	The username used to access this repository (if protected by HTTP Basic authentication)
`repos."name".password`	The password used to access this repository (if protected by HTTP Basic authentication)
`repos."name".skipSslValidation`	For a `https://` URI, whether to skip validation of the SSL certificate on the default repository’s server. Valid values are `true` and `false`
`repos."name".hostKey`	The host key used by the Config Server to access this repository (if accessing via SSH). See the SSH Repository Access section for more information
`repos."name".hostKeyAlgorithm`	The algorithm of `hostKey`: one of “ssh-dss”, “ssh-rsa”, “ecdsa-sha2-nistp256”, “ecdsa-sha2-nistp384”, and “ecdsa-sha2-nistp521”
`repos."name".privateKey`	The private key corresponding to `hostKey`, with all newline characters replaced by `\n`

Important: If you set `cloneOnStart` to `true` for a repository which is secured with HTTP Basic authentication, you must set the `username` and `password` at the same time as you set `cloneOnStart`. Otherwise, the Config Server will be unable to access the repository and the service instance may fail to initialize.

The `uri` setting is required; you cannot define a Config Server configuration source without including a `uri`.

The default value of the `label` setting is `master`. You can set `label` to a branch name, a tag name, or a specific Git commit hash.

To set `label` to point to the `develop` branch of a repository, you might configure the setting as shown in the following JSON:

```
'{"git": { "repos": { "cookie": { "uri": "https://github.com/myorg/config-repo", "label": "develop" } } } }'
```

To set `label` to point to the `v1.1` tag in a repository, you might configure the setting as shown in the following JSON:

```
'{"git": { "repos": { "cookie": { "uri": "https://github.com/myorg/config-repo", "label": "v1.1" } } } }'
```

Within a client application, you can override this `label` setting’s value by setting the `spring.cloud.config.label` property (for example, in `bootstrap.yml`).

```
spring:
  cloud:
    config:
      label: v1.2
```

Passwords are masked in the Config Server dashboard.

To configure a Config Server service instance with a default repository and a repository specific to an application named “cook”, use the following JSON object:

```
'{"git": { "uri": "https://github.com/spring-cloud-services-samples/fortune-teller", "searchPaths": "configuration", "repos": { "cookie": { "pattern": "cook", "uri": "https://github.com/spring-cloud-services-samples/cook-config" } } } }'
```

To configure a Config Server service instance with a default repository and a repository specific to applications using the `dev` profile, use the following JSON object:

```
'{"git": { "uri": "https://github.com/spring-cloud-services-samples/fortune-teller", "searchPaths": "configuration", "repos": { "cookie": { "pattern": "*/dev", "uri": "https://github.com/spring-cloud-services-samples/cook-config" } } } }'
```

Placeholders in Repository URIs

The URIs for configuration source Git repositories can include a couple of special strings as placeholders:

- `{application}`: the name set in the `spring.application.name` property on an application
- `{profile}`: a profile listed in the `spring.profiles.active` property on an application

You can use these placeholders to (for example) set a single URI which maps one repository each to multiple applications that use the same Config Server, or to set a single URI which maps one repository each to multiple profiles.

Note: URI placeholders cannot be used with a repository that has the `cloneOnStart` setting set to `true`. See the listing for `cloneOnStart` in the table of general configuration parameters.

A repository URI that enables use of one repository per application might be expressed as shown in the following JSON. For an application named "cook", this would locate the repository named `cook-config`:

```
'{"git": { "uri": "https://github.com/spring-cloud-services-samples/{application}-config" } }'
```

A repository URI that enables use of one repository per profile might be expressed as shown in the following JSON. For an application using the `dev` profile, this would locate a repository named `config-dev`:

```
'{"git": { "uri": "https://github.com/spring-cloud-services-samples/config-{profile}" } }'
```

For more information about using placeholders, see "Placeholders in Git URI" in the Spring Cloud Config documentation.

To configure a Config Server service instance with a repository URI that enables one repository per application, use the following JSON object:

```
'{"git": { "uri": "https://github.com/spring-cloud-services-samples/{application}-config" } }'
```

To configure a Config Server service instance with a repository URI that enables one repository per profile, use the following JSON object:

```
'{"git": { "uri": "https://github.com/spring-cloud-services-samples/config-{profile}" } }'
```

HTTP(S) Proxy Repository Access

You can configure a Config Server service instance to access configuration sources using an HTTP or HTTPS proxy. To do so, you must provide proxy settings in either of the `git.proxy.http` or `git.proxy.https` JSON objects. You can set the proxy host and port, the proxy username and password (if applicable), and a list of hosts which the Config Server should access outside of the proxy.

Note: Proxy settings must be set once in each `git` object, where applicable. If you are using a composite backend with multiple configuration sources that use the same proxy, you must provide that proxy's settings for each configuration source object.

Settings for an HTTP proxy are set in the `git.proxy.http` object. These might be set as shown in the following JSON:

```
'{"git": { "proxy": { "http": { "host": "proxy.wise.com", "port": "80" } } } }'
```

Settings for an HTTPS proxy are set in the `git.proxy.https` object. These might be set as shown in the following JSON:

```
'{"git": { "proxy": { "https": { "host": "secure.wise.com", "port": "443" } } } }'
```

Note: Some networks require that separate proxy servers are used for HTTP and HTTPS URLs. In such a case, you can set both the `proxy.http` and `proxy.https` objects.

The parameters used to configure HTTP or HTTPS proxy settings for the Config Server are listed below.

Parameter	Function
`proxy.http`	A proxy object, containing HTTP proxy fields
`proxy.http.host`	The HTTP proxy host
`proxy.http.port`	The HTTP proxy port
`proxy.http.nonProxyHosts`	The hosts to access outside the HTTP proxy
`proxy.http.username`	The username to use with an authenticated HTTP proxy
`proxy.http.password`	The password to use with an authenticated HTTP proxy
`proxy.https`	A proxy object, containing HTTPS proxy fields

Parameter	Function
`proxy.https.host`	The HTTPS proxy host
`proxy.https.port`	The HTTPS proxy port
`proxy.https.nonProxyHosts`	The hosts to access outside the HTTPS proxy (if `proxy.http.nonProxyHosts` is also provided, `http.nonProxyHosts` will be used instead of `https.nonProxyHosts`)
`proxy.https.username`	The username to use with an authenticated HTTPS proxy (if `proxy.http.username` is also provided, `http.username` will be used instead of `https.username`)
`proxy.https.password`	The password to use with an authenticated HTTPS proxy (if `proxy.http.password` is also provided, `http.password` will be used instead of `https.password`)

To configure a Config Server service instance that uses an HTTP proxy to access configuration sources, use the following JSON object:

```
'{"git": {"uri": "https://github.com/spring-cloud-services-samples/cook-config", "proxy": { "http": { "host": "proxy.wise.com", "port": "80" } } } }'
```

To configure a Config Server service instance that uses an authenticated HTTPS proxy to access configuration sources, specifying that `example.com` should be accessed outside of the proxy, use the following JSON object:

```
'{"git": {"uri": "https://github.com/spring-cloud-services-samples/cook-config", "proxy": { "https": { "host": "secure.wise.com", "port": "443", "username": "jim", "password": "wright62", "nonProxyHosts": "example.com" } } } }'
```

Configuring with Vault

Page last updated:

Overview

HashiCorp Vault is a secrets management tool, which encrypts and stores credentials, API keys, and other secrets for use in distributed systems. It provides support for access control lists, secret revocation, auditing, and leases and renewals, and includes special capabilities for common infrastructure and systems such as AWS, MySQL, and RabbitMQ, among others. For more information about Vault, see the documentation.

Spring Cloud Config provides a Vault backend so that the Spring Cloud Config Server can serve configuration stored in Vault. The Spring Cloud Services Config Server supports this backend and can serve configuration stored in Vault to client applications which have been given access to the Vault server (this includes provision of a Vault access token for the client application).

Important: Spring Cloud Services does not provide a HashiCorp Vault server. You must provide your own Vault server in order to use Config Server with Vault.

See below for information about configuring a Config Server service instance to use a HashiCorp Vault server for a configuration source.

General Configuration

Parameters used to configure a configuration source are part of a JSON object called `vault`, as in `{"vault": { "host": "127.0.0.1", "port": "8200" } }`.

Important: The Spring Cloud Services Config Server supports only one Vault backend, so only one `vault` object is permitted in the configuration parameters.

General parameters used to configure a Config Server configuration source are listed below.

Parameter	Function
`host`	The host of the Vault server
`port`	The port of the Vault server
`scheme`	The URI scheme used in accessing the Vault server (default value: `http`)
`backend`	The name of the Vault backend from which to retrieve configuration (default value: `secret`)
`defaultKey`	The default key from which to retrieve configuration (default value: `application`)
`profileSeparator`	The value used to separate profiles (default value: `,`)
`skipSslValidation`	Whether to skip validation of the SSL certificate on the Vault server. Valid values are `true` and `false`

The value of `defaultKey` is masked in the Config Server dashboard.

For information about writing a client application that accesses configuration values from a Config Server which has been configured to use Vault, see the Use a HashiCorp Vault Server section of the Writing Client Applications topic.

HTTP(S) Proxy Repository Access

You can configure a Config Server service instance to access a configuration source using an HTTP or HTTPS proxy. To do so, you must provide proxy settings in either of the `vault.proxy.http` or `vault.proxy.https` JSON objects. You can set the proxy host and port, the proxy username and password (if applicable), and a list of hosts which the Config Server should access outside of the proxy.

Settings for an HTTP proxy are set in the `vault.proxy.http` object. These might be set as shown in the following JSON:

```
'{"vault": { "proxy": { "http": { "host": "proxy.wise.com", "port": "80" } } } }'
```

Settings for an HTTPS proxy are set in the `vault.proxy.https` object. These might be set as shown in the following JSON:

```
'{"vault": { "proxy": { "https": { "host": "secure.wise.com", "port": "443" } } } }'
```

Note: Some networks require that separate proxy servers are used for HTTP and HTTPS URLs. In such a case, you can set both the `proxy.http` and `proxy.https` objects.

The parameters used to configure HTTP or HTTPS proxy settings for the Config Server are listed below.

Parameter	Function
`proxy.http`	A proxy object, containing HTTP proxy fields
`proxy.http.host`	The HTTP proxy host
`proxy.http.port`	The HTTP proxy port
`proxy.http.nonProxyHosts`	The hosts to access outside the HTTP proxy
`proxy.http.username`	The username to use with an authenticated HTTP proxy

Parameter	Function
`proxy.http.password`	The password to use with an authenticated HTTP proxy
`proxy.https`	A proxy object, containing HTTPS proxy fields
`proxy.https.host`	The HTTPS proxy host
`proxy.https.port`	The HTTPS proxy port
`proxy.https.nonProxyHosts`	The hosts to access outside the HTTPS proxy (if `proxy.http.nonProxyHosts` is also provided, `http.nonProxyHosts` will be used instead of `https.nonProxyHosts`)
`proxy.https.username`	The username to use with an authenticated HTTPS proxy (if `proxy.http.username` is also provided, `http.username` will be used instead of `https.username`)
`proxy.https.password`	The password to use with an authenticated HTTPS proxy (if `proxy.http.password` is also provided, `http.password` will be used instead of `https.password`)

To configure a Config Server service instance that uses an HTTP proxy to access a configuration source, use the following JSON object:

```
'{"vault": { "host": "127.0.0.1", "port": "8200", "proxy": { "http": { "host": "proxy.wise.com", "port": "80" } } } }'
```

To configure a Config Server service instance that uses an authenticated HTTPS proxy to access a configuration source, specifying that `example.com` should be accessed outside of the proxy, use the following JSON object:

```
'{"vault": { "host": "127.0.0.1", "port": "8200", "proxy": { "https": { "host": "secure.wise.com", "port": "443", "username": "jim", "password": "wright62", "nonProxyHosts": "example.com" } } } }'
```

Declarative Composite Backends

Page last updated:

Overview

The Spring Cloud Services Config Server provides the ability to serve configuration properties from a composite of multiple backends, such as from multiple GitHub repositories and a HashiCorp Vault server. This feature builds upon the Composite Environment Repositories feature from the Dalston release of the Spring Cloud Config Server by enabling declaration of multiple Git servers together with (optionally) a Vault server to compose a unified configuration source.

Note: The Spring Cloud Services Config Server cannot serve configuration properties from multiple Vault servers. A composite backend can contain only a single Vault server.

Property Source Precedence

When a Config Server is composed with multiple backends, a request from a client application retrieves properties found in any applicable backend within the composite. The composite backends are always searched in the order in which they are configured, and the Config Server assembles configuration properties beginning at the first backend in the composite by order.

For example, given the following configuration:

- Property `dbhost=foo` is defined in the first backend
- Property `dbhost=bar` is defined in the second backend
- Property `port=4321` is defined in the third backend

The client will resolve property `dbhost` to `foo`, from the first backend, as that backend has precedence over the second. Property `port` will be resolved to `4321`, as it is only defined in the third backend.

Fail-Fast Behavior

If the Config Server encounters an error when retrieving properties from any one of the backends, the client application's request fails. The Config Server's composite configuration sources will only be available once all backends are available.

Use of Labels

Within the composite, all repositories or servers must contain the same labels—branches or tags in a Git repository, or paths in a Vault server. A label in a Config Server configuration source corresponds to a Spring application profile on a client application. If a client application makes a request for configuration for a given profile and one of the composite's backends does not have a corresponding branch, tag, or path, the request fails and no configuration is returned.

See below for information about configuring a Config Server service instance to use a composite backend for configuration sources.

General Configuration

Parameters used to configure configuration sources are part of a JSON array called `composite`. The `composite` array can contain one or more `git` JSON objects, each of which contains settings for a Git repository, and a `vault` object, which contains settings for a Vault server. For information about configuring a Git configuration source, see the Configuring with Git topic. For information about configuring a Vault configuration source, see the Configuring with Vault topic.

In a composite backend, a Git configuration source can use the additional properties described below.

Parameter	Function
`git.pattern`	A pattern for the names of applications that store configuration in the repository
`git.searchPaths`	A pattern used to search for configuration-containing subdirectories in the repository

Configuration properties from individual backends are given precedence based on the order in which they are provided to the Config Server. A backend specified later in the `composite` array is searched after backends specified earlier in the array.

To configure a Config Server service instance to use a composite backend comprising two Git repositories and a Vault server, you might use the following JSON object:

```
'{
  "composite": [
    {
      "git": {
        "uri": "https://github.com/spring-cloud-services-samples/cook-config",
        "pattern": "cook*"
      }
    },
    {
      "git": {
        "uri": "https://github.com/spring-cloud-samples/config-repo",
        "searchPaths": "specialConfig"
      }
    },
    {
      "vault": {
        "host": "127.0.0.1",
        "port": 8200,
        "scheme": "https",
        "backend": "secret",
        "defaultKey": "application",
        "profileSeparator": ","
      }
    }
  ]
}'
```

In this example, configuration properties found in the Vault server are added to the response only if they are not found in either of the Git repositories specified before the Vault server.

To configure a Config Server service instance to use a composite backend comprising a Vault server and a Git repository, use the following JSON object:

```
'{
  "composite": [
    {
      "vault": {
        "host": "127.0.0.1",
        "port": 8200,
        "scheme": "https",
        "backend": "secret",
        "defaultKey": "application",
        "profileSeparator": ","
      }
    },
    {
      "git": {
        "uri": "https://github.com/spring-cloud-services-samples/cook-config"
      }
    }
  ]
}'
```

In this example, the response consists of configuration properties found in the Vault server, plus any properties found only in the Git repository.

For information about using an application to access configuration values served by a Config Server service instance, see the Writing Client Applications topic.

Proxy Servers for Composite Backends

Proxy server configuration must be defined for each item in the composite set. For example:

```
'{
  "composite": [
    {
      "git": {
        "uri": "https://github.com/spring-cloud-services-samples/cook-config",
        "proxy": {
          "http": {
            "host": "http://proxy1.com",
            "port": 3218
          }
        }
      }
    },
    {
      "git": {
        "uri": "https://github.com/spring-cloud-samples/config-repo",
        "proxy": {
          "http": {
            "host": "http://proxy2.com",
            "port": 3218
          }
        }
      }
    },
    {
      "vault": {
        "host": "127.0.0.1",
        "proxy": {
          "http": {
            "host": "http://proxy1.com",
            "port": 3218
          }
        }

      }
    }
  ]
}'
```

For detailed information about configuring proxy support, see the HTTP(S) Proxy Repository Access section of the Configuring with Git topic and the HTTP(S) Proxy Repository Access section of the Configuring with Vault topic.

Configuration Properties

Page last updated:

The Config Server can serve configuration properties from either Git or HashiCorp Vault configuration sources. Configuration properties can be applicable to all applications that use the Config Server, specific to an application, or specific to a Spring application profile, and can be stored in encrypted form.

See below for information about the structure and format of configuration properties to be served by the Config Server.

Git

If using a Git configuration source, you must store properties in YAML or Java .properties files.

Global Configuration

You can store configuration properties so that they are served to all applications which use the Config Server. In the configuration repository, a file named `application.yml` or `application.properties` contains configuration which will be served to all applications that access the Config Server.

Using YAML

An example of a global `application.yml` file:

```
message: Hi there!
```

Using a Properties File

An example of a global `application.properties` file:

```
message=Hi there!
```

Application-Specific Configuration

You can store configuration properties so that they are served only to a specific application. In the configuration repository, a file named `[APP-NAME].yml` or `[APP-NAME].properties`, where `[APP-NAME]` is the name of an application, contains configuration which will be served only to the `APP-NAME` application.

Using YAML

An example of an application-specific `cook.yml` file:

```
server:
  port: 80

cook:
  special: Fried Salamander
```

Using a Properties File

An example of an application-specific `cook.properties` file:

```
server.port=80

cook.special=Fried Salamander
```

Profile-Specific Configuration

You can store configuration properties so that they are served only to applications which have activated a specific Spring application profile. In the configuration repository, a file named `[APP-NAME]-[PROFILE-NAME].yml` or `[APP-NAME]-[PROFILE-NAME].properties`, where `[APP-NAME]` is the name of an application and `[PROFILE-NAME]` is the name of an application profile, contains configuration which will be served only to the `APP-NAME` application running with the `[PROFILE-NAME]` profile activated. Within a YAML file named `[APP-NAME].yml`, a document that begins by setting the `spring.profiles` property contains configuration which will be served only to the `APP-NAME` application running with the profile specified by the `spring.profiles` property.

Using YAML

An example of a profile-specific `cook-dev.yml` file:

```
server:
  port: 8080

cook:
  special: Birdfeather Tea
```

An example of a profile-specific YAML document within a `cook.yml` file:

```
---
spring:
  profiles: dev

server:
  port: 8080

cook:
  special: Birdfeather Tea
```

Using a Properties File

An example of a profile-specific `cook-dev.properties` file:

```
server.port=8080

cook.special=Birdfeather Tea
```

Encrypted Configuration

You can store configuration properties in encrypted form and have these properties decrypted by the Config Server before they are served to applications. In a file within the configuration repository, properties whose values are prefixed with `{cipher}` will be decrypted before they are served to client applications. To use this feature, you must configure the Config Server with an encryption key as described in the Encryption and Encrypted Values section of the Configuring with Git topic.

Using YAML

In a YAML file, an encrypted property value must be surrounded by single quotes.

An example of an encrypted property value in an `application.yml` file:

```
secretMenu: '{cipher}AQA90Q3GIRAMu6ToMqwS++En2iFzMXIWX99G66yaZFRHrQNq64CntqOzWymd3xE7uJp
ZKQc9XBIkfyRz/HUGhXRdf3KZQ9bqclwmR5vkiLmN9DHlAxS+6biT+7f8ptKo3fzQ0gGOBaR4kTnWLBxmVaIkjq1
Qze4aIgsgUWuhbEek+3znkH9+Mc+5zNPvwN8hhgDMDVzgZLB+4YnvWJAq3Au4wEevakAHHxVY0mXcxj1Ro+H+Zel
IzfF8K2AvC3vmvlmxy9Y49Zjx0RhMzUx17eh3mAB8UMMRJZyUG2a2uGCXmz+UunTA5n/dWWOvR3VcZyzXPFSFkhN
ekw3db9XZ7goceJSPrRN+5s+GjLCPr+KSnhLmUt1XAScMeqTieNCHT5I='
```

Using a Properties File

An example of an encrypted property value in an `application.properties` file:

```
secretMenu={cipher}AQA90Q3GIRAMu6ToMqwS++En2iFzMXIWX99G66yaZFRHrQNq64CntqOzWymd3xE7uJpZK
Qc9XBIkfyRz/HUGhXRdf3KZQ9bqclwmR5vkiLmN9DHlAxS+6biT+7f8ptKo3fzQ0gGOBaR4kTnWLBxmVaIkjq1Qz
e4aIgsgUWuhbEek+3znkH9+Mc+5zNPvwN8hhgDMDVzgZLB+4YnvWJAq3Au4wEevakAHHxVY0mXcxj1Ro+H+ZelIz
fF8K2AvC3vmvlmxy9Y49Zjx0RhMzUx17eh3mAB8UMMRJZyUG2a2uGCXmz+UunTA5n/dWWOvR3VcZyzXPFSFkhNek
w3db9XZ7goceJSPrRN+5s+GjLCPr+KSnhLmUt1XAScMeqTieNCHT5I=
```

Vault

If using a HashiCorp Vault configuration source, you must write secrets to the Vault server using the `vault` Command Line Interface (CLI) tool.

Global Configuration

You can store configuration properties so that they are served to all applications which use the Config Server. A secret written to the `secret/application` path will be served to all applications that access the Config Server.

An example of setting a global configuration property:

```
$ vault write secret/application message=Greetings
```

Application-Specific Configuration

You can store configuration properties so that they are served only to a specific application. A secret written to the `secret/[APP-NAME]` path contains configuration which will be served only to the `APP-NAME` application.

An example of setting an application-specific configuration property:

```
$ vault write secret/cook message=Hi
```

Profile-Specific Configuration

You can store configuration properties so that they are served only to applications which have activated a specific Spring application profile. A secret written to the `secret/[APP-NAME],[PROFILE-NAME]` path, where `[APP-NAME]` is the name of an application and `[PROFILE-NAME]` is the name of an application profile, contains configuration which will be served only to the `APP-NAME` application running with the `[PROFILE-NAME]` profile activated.

An example of setting a profile-specific configuration property:

```
$ vault write secret/cook,dev message=Ho
```

Writing Client Applications

Page last updated:

Refer to the "Cook" sample application to follow along with the code in this topic.

To use a Spring Boot application as a client for a Config Server instance, you must add the dependencies listed in the Client Dependencies topic to your application's build file. Be sure to include the dependencies for Config Server as well.

Important: Because of a dependency on Spring Security, the Spring Cloud® Config Client starter will by default cause all application endpoints to be protected by HTTP Basic authentication. If you wish to disable this, please see Disable HTTP Basic Authentication below.

Add Self-Signed SSL Certificate to JVM Truststore

Spring Cloud Services uses HTTPS for all client-to-service communication. If your Pivotal Cloud Foundry installation is using a self-signed SSL certificate, the certificate will need to be added to the JVM truststore before your client application can consume properties from a Config Server service instance.

Spring Cloud Services can add the certificate for you automatically. For this to work, you must set the `TRUST_CERTS` environment variable on your client application to the API endpoint of your Elastic Runtime instance:

```
$ cf set-env cook TRUST_CERTS api.cf.wise.com
Setting env variable 'TRUST_CERTS' to 'api.cf.wise.com' for app cook in org myorg / space development as user...
OK
TIP: Use 'cf restage' to ensure your env variable changes take effect

$ cf restage cook
```

Note: The `CF_TARGET` environment variable was formerly recommended for configuring Spring Cloud Services to add a certificate to the truststore. `CF_TARGET` is still supported for this purpose, but `TRUST_CERTS` is more flexible and is now recommended instead.

As the output from the `cf set-env` command suggests, restage the application after setting the environment variable.

Use Configuration Values

When the application requests a configuration from the Config Server, it will use a path containing the application name (as described in the Configuration Clients section of the Background Information topic). You can declare the application name in `bootstrap.properties`, `bootstrap.yml`, `application.properties`, or `application.yml`.

In `bootstrap.yml`:

```
spring:
  application:
    name: cook
```

This application will use a path with the application name `cook`, so the Config Server will look in its configuration source for files whose names begin with `cook`, and return configuration properties from those files.

Now you can (for example) inject a configuration property value using the `@Value` annotation. The Menu class reads the value of `special` from the `cook.special` configuration property.

```
@RefreshScope
@Component
public class Menu {

  @Value("${cook.special}")
  String special;

  //...

  public String getSpecial() {
    return special;
  }

  //...

}
```

The `Application` class is a `@RestController`. It has an injected `menu` and returns the `special` (the value of which will be supplied by the Config Server) in its `restaurant()` method, which it maps to `/restaurant`.

```
@RestController
@SpringBootApplication
public class Application {

  @Autowired
  private Menu menu;

  @RequestMapping("/restaurant")
  public String restaurant() {
    return String.format("Today's special is: %s", menu.getSpecial());
  }
  //...
```

Vary Configurations Based on Profiles

You can provide configurations for multiple profiles by including appropriately-named `.yml` or `.properties` files in the Config Server instance's configuration source (the Git repository). Filenames follow the format `{application}-{profile}.{extension}`, as in `cook-production.yml`. (See the The Config Server section of the Background Information topic.)

The application will request configurations for any active profiles. To set profiles as active, you can use the `SPRING_PROFILES_ACTIVE` environment variable, set for example in `manifest.yml`.

```
applications:
  - name: cook
    host: cookie
    services:
      - config-server
    env:
      SPRING_PROFILES_ACTIVE: production
```

The sample configuration source cook-config contains the files `cook.properties` and `cook-production.properties`. With the active profile set to `production` as in `manifest.yml` above, the application will make a request of the Config Server using the path `/cook/production`, and the Config Server will return properties from both `cook-production.properties` (the profile-specific configuration) and `cook.properties` (the default configuration); for example:

```
{
  "name":"cook",
  "profiles":[
    "production"
  ],
  "label":"master",
  "propertySources":[
    {
      "name":"https://github.com/spring-cloud-services-samples/cook-config/cook-production.properties",
      "source":
        {
          "cook.special":"Cake a la mode"
        }
    },
    {
      "name":"https://github.com/spring-cloud-services-samples/cook-config/cook.properties",
      "source":
        {
          "cook.special":"Pickled Cactus"
        }
    }
  ]
}
```

As noted in the Configuration Clients section of the Background Information topic, the application must decide what to do when the server returns multiple values for a configuration property, but a Spring application will take the first value for each property. In the example response above, the configuration for the specified profile (`production`) is first in the list, so the Boot sample application will use values from that configuration.

View Client Application Configuration

Spring Boot Actuator adds an `env` endpoint to the application and maps it to `/env`. This endpoint displays the application's profiles and property sources from the Spring `ConfigurableEnvironment`. (See "Endpoints" in the "Spring Boot Actuator" section of the Spring Boot Reference Guide.) In the case of an application which is bound to a Config Server service instance, `env` will display properties provided by the instance.

To use Actuator, you must add the `spring-boot-starter-actuator` dependency to your project. If using Maven, add to `pom.xml`:

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

If using Gradle, add to `build.gradle`:

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

You can now visit `/env` to see the application environment's properties (the following shows an excerpt of an example response):

```
$ curl http://cookie.apps.wise.com/env
{
  "profiles":[
    "dev","cloud"
  ],
  "configService:https://github.com/spring-cloud-services-samples/cook-config/cook.properties":{
    "cook.special":"Pickled Cactus"
  },
  "vcap":{
    "vcap.application.limits.mem":"512",
    "vcap.application.application_uris":"cookie.apps.wise.com",
    "vcap.services.config-server.name":"config-server",
    "vcap.application.uris":"cookie.apps.wise.com",
    "vcap.application.application_version":"179de3f9-38b6-4939-bff5-41a14ce4e700",
    "vcap.services.config-server.tags[0]":"configuration",
    "vcap.application.space_name":"development",
    "vcap.services.config-server.plan":"standard",
//...
```

Refresh Client Application Configuration

Spring Boot Actuator also adds a `refresh` endpoint to the application. This endpoint is mapped to `/refresh`, and a POST request to the `refresh` endpoint refreshes any beans which are annotated with `@RefreshScope`. You can thus use `@RefreshScope` to refresh properties which were initialized with values provided by the Config Server.

The `Menu.java` class is marked as a `@Component` and also annotated with `@RefreshScope`.

```
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.stereotype.Component;

@RefreshScope
@Component
public class Menu {

  @Value("${cook.special}")
  String special;
  //...
```

This means that after you change values in the configuration source repository, you can update the `special` on the `Application` class's `menu` with a refresh event triggered on the application:

```
$ curl http://cookie.apps.wise.com/restaurant
Today's special is: Pickled Cactus

$ git commit -am "new special"
[master 3c9ff23] new special
 1 file changed, 1 insertion(+), 1 deletion(-)

$ git push

$ curl -X POST http://cookie.apps.wise.com/refresh
["cook.special"]

$ curl http://cookie.apps.wise.com/restaurant
Today's special is: Birdfeather Tea
```

Use Client-Side Decryption

On the Config Server, the decryption features are disabled, so encrypted property values from a configuration source are delivered to client applications unmodified. You can use the decryption features of Spring Cloud Config Client to perform client-side decryption of encrypted values.

To use the decryption features in a client application, you must use a Java buildpack which contains the Java Cryptography Extension (JCE) Unlimited Strength policy files. These files are contained in the Cloud Foundry Java buildpack from version 3.7.1.

> If you cannot use version 3.7.1 or later, you can add the JCE Unlimited Strength policy files to an earlier version of the Cloud Foundry Java buildpack. Fork the buildpack on GitHub, then download the policy files from Oracle and place them in the buildpack's `resources/open_jdk_jre/lib/security` directory. Follow the instructions in the Managing Custom Buildpacks topic to add this buildpack to Pivotal Cloud Foundry. Be sure that it has the lowest position of all enabled Java buildpacks.

You must also include Spring Security RSA as a dependency.

If using Maven, include in `pom.xml`:

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-rsa</artifactId>
</dependency>
```

If using Gradle, include in `build.gradle`:

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

Encrypted values must be prefixed with the string `{cipher}`. If using YAML, enclose the entire value in single quotes, as in `'{cipher}vALuE'`; if using a properties file, do not use quotes. The configuration source for the Cook application has a `secretMenu` property in its `cook-encryption.properties`:

```
secretMenu={cipher}AQA90Q3GIRAMu6ToMqwS++En2iFzMXIWX99G66yaZFRHrQNq64CntqOzWymd3xE7uJpZK
Qc9XBIkfyRz/HUGhXRdf3KZQ9bqclwmR5vkiLmN9DHIAxS+6biT+7f8ptKo3fzQ0gGOBaR4kTnWLBxmVaIkjq1Qz
e4aIgsgUWuhbEek+3znkH9+Mc+5zNPvwN8hhgDMDVzgZLB+4YnvWJAq3Au4wEevakAHHxVY0mXcxj1Ro+H+ZelIz
fF8K2AvC3vmvlmxy9Y49Zjx0RhMzUx17eh3mAB8UMMRJZyUG2a2uGCXmz+UunTA5n/dWWOvR3VcZyzXPFSFkhNek
w3db9XZ7goceJSPrRN+5s+GjLCPr+KSnhLmUt1XAScMeqTieNCHT5I=
```

Put the key on the client application classpath. You can either add the keystore to the buildpack's `resources/open_jdk_jre/lib/security` directory (as described above for the JCE policy files) or include it with the source for the application. In the Cook application, the key is placed in `src/main/resources`.

```
cook
└── src
    └── main
        └── resources
            ├── application.yml
            ├── bootstrap.yml
            └── server.jks
```

Important: Cook is an example application, and the key is packaged with the application source for example purposes. If at all possible, use the

buildpack for keystore distribution.

Specify the key location and credentials in `bootstrap.properties` or `bootstrap.yml` using the `encrypt.keyStore` properties:

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

The Menu class has a String property `secretMenu`.

```
@Value("${secretMenu}")
String secretMenu;

//...

public String getSecretMenu() {
  return secretMenu;
}
```

A default value for `secretMenu` is in `bootstrap.yml`:

```
secretMenu: Animal Crackers
```

In the Application class, the method `secretMenu()` is mapped to `/restaurant/secret-menu`. It returns the value of the `secretMenu` property.

```
@RequestMapping("/restaurant/secret-menu")
public String secretMenu() {
  return menu.getSecretMenu();
}
```

After making the key available to the application and installing the JCE policy files in the Java buildpack, you can cause the Config Server to serve properties from the `cook-encryption.properties` file by activating the `encryption` profile on the application, e.g. by running `cf set-env` to set the `SPRING_PROFILES_ACTIVE` environment variable:

```
$ cf set-env cook SPRING_PROFILES_ACTIVE dev,encryption
Setting env variable 'SPRING_PROFILES_ACTIVE' to 'dev,encryption' for app cook in org myorg / space development as user...
OK
TIP: Use 'cf restage' to ensure your env variable changes take effect

$ cf restage cook
```

The application will decrypt the encrypted property after receiving it from the Config Server. You can view the property value by visiting `/restaurant/secret-menu` on the application.

Use a HashiCorp Vault Server

You can configure the Config Server to use a HashiCorp Vault server as a configuration source, as described in the Configuring with Vault topic. To consume configuration from the Vault server via the service instance, your client application must be given a Vault token. You can give the token to an application by setting the `SPRING_CLOUD_CONFIG_TOKEN` environment variable on the application. The Spring Cloud Services Connectors for Config Server will automatically renew the application's token for as long as the application is running.

Important: If the application is entirely stopped (i.e., no instances continue to run) and its Vault token expires, you will need to create a new token for the application and re-set the `SPRING_CLOUD_CONFIG_TOKEN` environment variable.

To generate a token for use in the application, you can run the `vault token-create` command, providing a Time To Live (TTL) that is long enough for the application to be restaged after you have set the environment variable. The following command creates a token with a TTL of one hour:

```
$ vault token-create -ttl="1h"
```

After generating the token, set the environment variable on your client application and then restage the application for the environment variable setting to take effect:

```
$ cf set-env cook SPRING_CLOUD_CONFIG_TOKEN c3432ef5-6a78-8673-ea23-5528c26849e4
Setting env variable 'SPRING_CLOUD_CONFIG_TOKEN' to 'c3432ef5-6a78-8673-ea23-5528c26849e4' for app cook in org myorg / space development as user...
OK
TIP: Use 'cf restage cook' to ensure your env variable changes take effect

$ cf restage cook
```

The Spring Cloud Services Connectors for Config Server renew the application token for as long as the application continues to run. For more information about the token renewal performed by the connectors, see the HashiCorp Vault Token Renewal section of the Spring Cloud Connectors topic.

Renew Vault Token Manually

After creating a Vault token for an application, you can renew the token manually using the Config Server service instance bound to the application.

Note: The following procedure uses the jq command-line JSON processing tool.

Run `cf env`, giving the name of an application that is bound to the service instance:

```
$ cf services
Getting services in org myorg / space development as admin...
OK

name            service          plan       bound apps   last operation
config-server   p-config-server  standard   vault-app    create succeeded

$ cf env vault-app
Getting env variables for app vault-app in org myorg / space development as admin...
OK

System-Provided:
{
 "VCAP_SERVICES": {
  "p-config-server": [
   {
    "credentials": {
     "access_token_uri": "https://p-spring-cloud-services.uaa.cf.wise.com/oauth/token",
     "client_id": "p-config-server-876cd13b-1564-4a9a-9d44-c7c8a6257b73",
     "client_secret": "rU7dMUw6bQjR",
     "uri": "https://config-86b38ce0-eed8-4c01-adb4-1a651a6178e2.apps.wise.com"
    },
[...]
```

Then create a Bash script that accesses the Vault token renewal endpoint on the service instance backing application. In the following two commands:

```
TOKEN=$(curl -k [ACCESS_TOKEN_URI] -u [CLIENT_ID]:[CLIENT_SECRET] -d
grant_type=client_credentials | jq -r .access_token); curl -H "Authorization: bearer
$TOKEN" -H "X-VAULT-Token: [VAULT_TOKEN]" -H "Content-Type: application/json" -X POST
[URI]/vault/v1/auth/token/renew-self -d '{"increment": [INTERVAL]}'
```

Replace the following placeholders using values from the `cf env` command above:

- `[ACCESS_TOKEN_URI]` with the value of `credentials.access_token_uri`
- `[CLIENT_ID]` with the value of `credentials.client_id`
- `[CLIENT_SECRET]` with the value of `credentials.client_secret`
- `[URI]` with the value of `credentials.uri`

Replace the following placeholders with the relevant values:

- `[VAULT_TOKEN]` with the Vault token string
- `[INTERVAL]` with the number of seconds to set as the Vault token's Time To Live (TTL)

After renewing the token, you can view its TTL by looking it up using the Vault command line. Run `vault token-lookup [TOKEN]`, replacing `[TOKEN]` with the Vault token string:

```
$ vault token-lookup 72ec7ca0-de41-b2dc-8fe4-d74c4c9a4e75
Key                 Value
---                 -----
accessor            436db91b-6bfb-9eec-7bfb-913260488ce8
creation_time       1493360487
creation_ttl        3600
display_name        token
explicit_max_ttl    0
id                  72ec7ca0-de41-b2dc-8fe4-d74c4c9a4e75
last_renewal_time   1493360718
meta
num_uses            0
orphan              false
path                auth/token/create
policies            [root]
renewable           true
ttl                 997
```

Disable HTTP Basic Authentication

The Spring Cloud Config Client starter has a dependency on Spring Security. Unless your application has other security configuration, this will cause all application endpoints to be protected by HTTP Basic authentication.

If you do not yet want to address application security, you can turn off Basic authentication by setting the `security.basic.enabled` property to `false`. In `application.yml` or `bootstrap.yml`:

```
security:
  basic:
    enabled: false
```

You might make this setting specific to a profile (such as the `dev` profile if you want Basic authentication disabled only for development):

```
---
spring:
  profiles: dev

security:
  basic:
    enabled: false
```

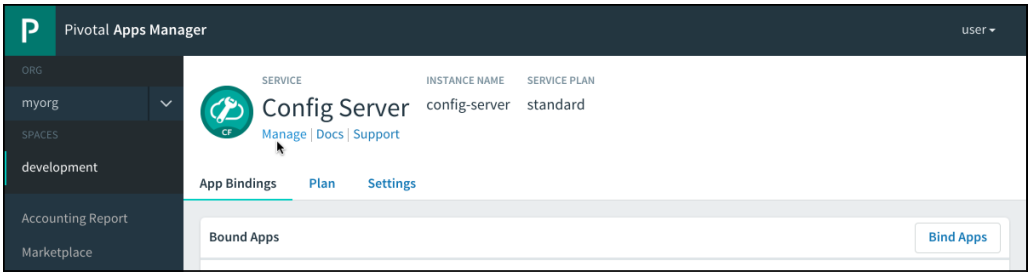
For more information, see "Security" in the Spring Boot Reference Guide.

Note: Because of the Spring Security dependency, HTTPS Basic authentication will also be enabled for Spring Boot Actuator endpoints. If you wish to disable that as well, you must also set the `management.security.enabled` property to `false`. See "Customizing the management server port" in the Spring Boot Reference Guide.

Using the Dashboard

Page last updated:

To find the dashboard, navigate in Pivotal Cloud Foundry® Apps Manager to the Config Server service instance's space, click the listing for the service instance, and then click **Manage**.

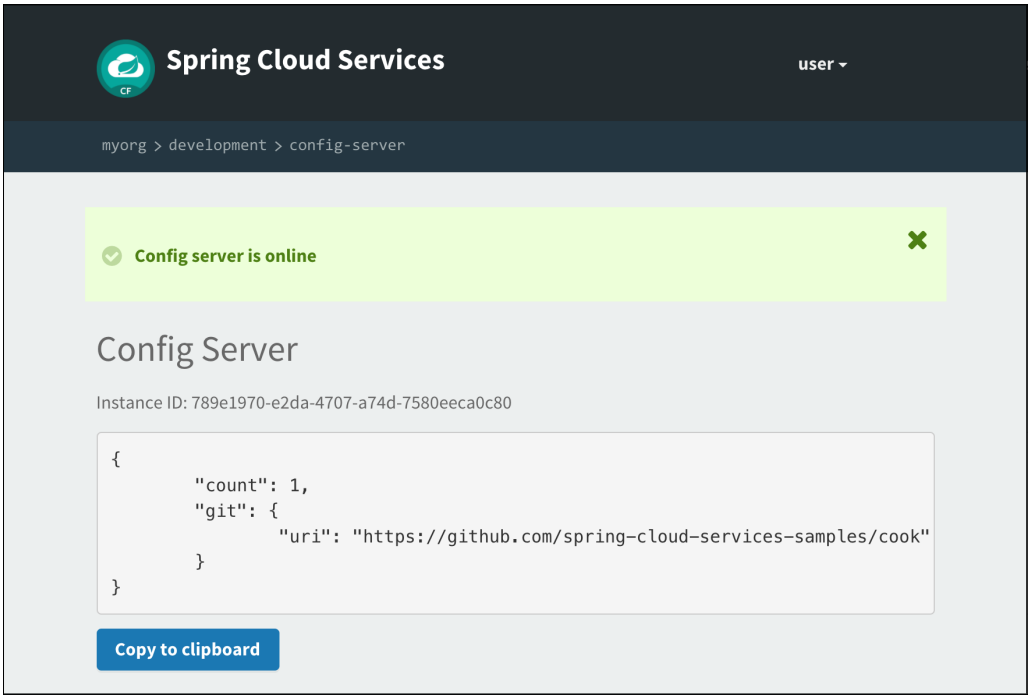


If you are using version 6.8.0 or later of the Cloud Foundry Command Line Interface tool (cf CLI), you can also use `cf service SERVICE_NAME`, where `SERVICE_NAME` is the name of the Config Server service instance:

```
$ cf service config-server

Service instance: config-server
Service: p-config-server
Bound apps:
Tags:
Plan: standard
Description: Config Server for Spring Cloud Applications
Documentation url: http://docs.pivotal.io/spring-cloud-services/
Dashboard: https://spring-cloud-broker.apps.wise.com/dashboard/p-config-server/650fd967-4e8e-4590-81a0-a029866184a2

Last Operation
Status: update succeeded
Message:
Started: 2016-06-27T14:30:23Z
Updated: 2016-06-27T14:32:30Z
```

Visit the URL given for "Dashboard".

The dashboard shows the current health of the Config Server along with the JSON object used to configure its settings. You can click the **Copy to clipboard** button to obtain this object in a string format suitable for use in command lines.

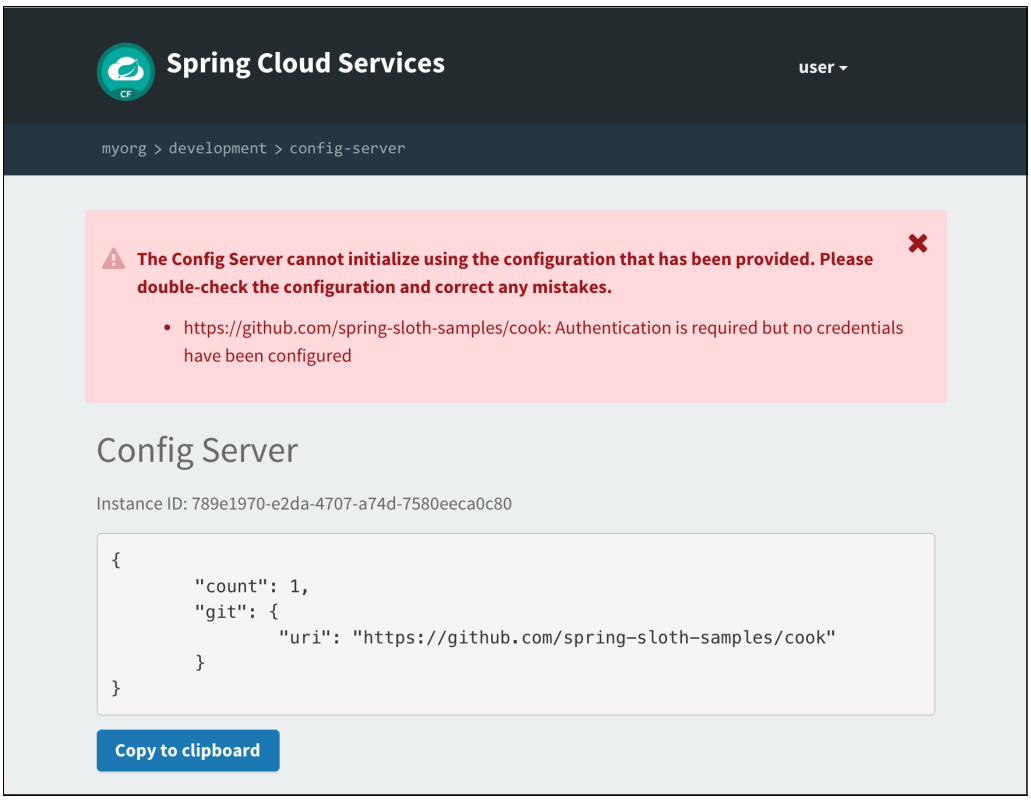


For the settings shown in the above figure, the button copies the following to the clipboard:

```
"{\"count\":1,\"git\":{\"privateKey\":\"****\",\"hostKey\":\"****\",\"hostKeyAlgorithm\":\"ssh-rsa\",\"uri\":\"git@github.com:spring-cloud-services-samples/cook.git\"},\"encrypt\":{\"key\":\"**
```

Important: If the Config Server service instance has been configured to use any credentials or encryption keys, you will need to edit the text provided by the **Copy to clipboard** button and replace the masked values (`****`) with the actual values.

If the Config Server encounters an error in attempting to load configuration from the configuration source or is otherwise unable to use the provided connection information, its dashboard will provide problem details, as shown in the example below.



Spring Cloud® Connectors

Page last updated:

To connect client applications to the Config Server, Spring Cloud Services uses Spring Cloud Connectors, including the Spring Cloud Cloud Foundry Connector, which discovers services bound to applications running in Cloud Foundry.

Service Detection

The connector inspects Cloud Foundry's `VCAP_SERVICES` environment variable, which stores connection and identification information for service instances that are bound to Cloud Foundry applications, to detect available services. Below is an example of a `VCAP_SERVICES` entry for the Spring Cloud Services Config Server.

```
{
 "VCAP_SERVICES": {
  "p-config-server": [
   {
    "credentials": {
     "access_token_uri": "https://p-spring-cloud-services.uaa.cf.wise.com/oauth/token",
     "client_id": "p-config-server-c4a56a3d-9507-4c2f-9cd1-f858dbf9e11c",
     "client_secret": "9aGx9K5Vx0cM",
     "uri": "https://config-51711835-4626-4823-b5a1-e5d91012f3f2.wise.com"
    },
    "label": "p-config-server",
    "name": "config-server",
    "plan": "standard",
    "tags": [
     "configuration",
     "spring-cloud"
    ]
   }
  ]
```

For each service in the `VCAP_SERVICES` variable, the connector considers the following fields:

- `tags` : Attributes or names of backing technologies behind the service.
- `label` : The service offering's name (not to be confused with a service *instance's* name).
- `credentials.uri` : A URI pertaining to the service instance.
- `credentials.uris` : URIs pertaining to the service instance.

Config Server Detection Criteria

To establish availability of the Config Server, the Spring Cloud Cloud Foundry Connector compares `VCAP_SERVICES` service entries against the following criteria:

- `tags` including `configuration`

Application Configuration

When the connector detects a Config Server service instance which has been bound to the application, it will automatically set the `spring.cloud.config.uri` property in the client application's environment, using the URL provided in the Config Server instance's `credentials` object. The connector will also set additional security properties to allow the client application to access the Config Server service instance.

HashiCorp Vault Token Renewal

When the `spring.cloud.config.token` property is set on an application, the connector enables automatic token renewal for a HashiCorp Vault client token. By default, the token's Time To Live (TTL) is set at 300000 milliseconds and the connector automatically renews the application's token every 60000 milliseconds.

You can configure the token's TTL and the renewal interval using properties under `vault.token` , set in the `application.yml` or `application.properties` file. The TTL is set using the `vault.token.ttl` property and the renewal interval is set using the `vault.token.renew.rate` property.

The following YAML sets the token TTL to 600000 milliseconds (10 minutes) and sets the renewal interval to 180000 milliseconds (three minutes).

```
vault:
  token:
    ttl: 600000
    renew:
      rate: 180000
```

HashiCorp Vault Property Source Redaction

In the Spring Boot Actuator `/env` endpoint, property names and values from a HashiCorp Vault property source are redacted for security. By default, the connector redacts properties whose names begin with `configService:vault:` , and they are displayed in the output of `/env` as shown in the following example:

```
"configService:vault:game": {
   "******": "Properties from this source are redacted for security reasons"
},
```

You can configure the pattern used to determine which properties to redact, as well as the message displayed in place of those properties' values, using properties set in the `application.yml` or `application.properties` file. The pattern is set using the `endpoints.env.mask.sourceNamePatterns` property (wildcards, written as `*`, are allowed) and the message is set using the `endpoints.env.mask.message` property.

For example, the following YAML sets the redacted property name pattern to match property sources whose names begin with `configService:` and sets the message to `Redacted for security`.

```
endpoints:
  env:
    mask:
      sourceNamePatterns: "configService:"
      message: "Redacted for security"
```

See Also

For more information about Spring Cloud Connectors, see the following:

- Spring Cloud Cloud Foundry Connector documentation
- Spring Cloud Connectors documentation
- Spring Cloud Connectors for Spring Cloud Services on Pivotal Cloud Foundry

Additional Resources

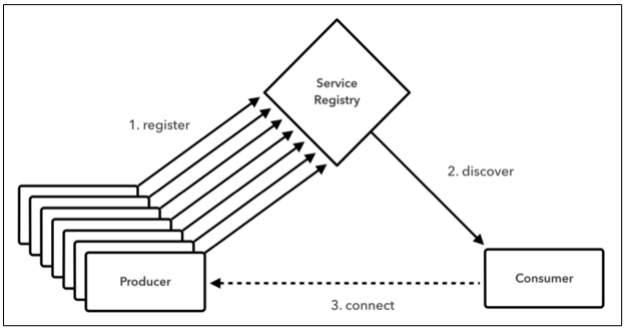
Page last updated:

- Spring Cloud® Services 1.0.0 on Pivotal Cloud Foundry - Config Server - YouTube (short screencast demonstrating Config Server for Pivotal Cloud Foundry)
- Spring Cloud Config (documentation for Spring Cloud Config, the open-source project that underlies Config Server for Pivotal Cloud Foundry)
- "Configuring It All Out" or "12-Factor App-Style Configuration with Spring" (blog post that provides background on Spring's configuration mechanisms and on Spring Cloud Config)
- Environment abstraction (Spring Framework Reference Documentation) (Spring Framework documentation on the `Environment` and the concepts of profiles and properties)

Service Registry for Pivotal Cloud Foundry

Overview

Service Registry for Pivotal Cloud Foundry (PCF) provides your applications with an implementation of the Service Discovery pattern, one of the key tenets of a microservice-based architecture. Trying to hand-configure each client of a service or adopt some form of access convention can be difficult and prove to be brittle in production. Instead, your applications can use the Service Registry to dynamically discover and call registered services.



When a client registers with the Service Registry, it provides metadata about itself, such as its host and port. The Registry expects a regular heartbeat message from each service instance. If an instance begins to consistently fail to send the heartbeat, the Service Registry will remove the instance from its registry.

Service Registry for Pivotal Cloud Foundry is based on Eureka, Netflix's Service Discovery server and client. For more information about Eureka and about the Service Discovery pattern, see Additional Resources.

Refer to the sample applications in the "greeting" repository to follow along with code in this section.

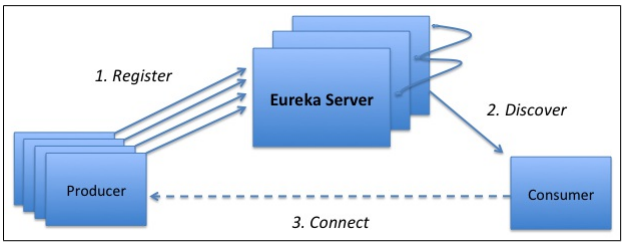
Open-Source Usage

Page last updated:

The Spring Cloud Services Service Registry is based on the Spring Cloud Netflix Eureka project. See below for information about the open-source Spring Cloud project.

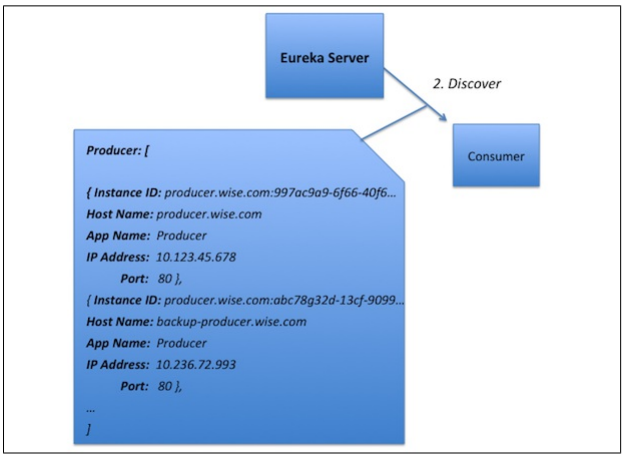
Open-Source Foundation: Spring Cloud Netflix Eureka

The Spring Cloud Netflix Eureka project, like the rest of Spring Cloud, builds on Spring Boot. It provides a wrapper around Eureka, Netflix's open-sourced service registry.



The servers in a Eureka cluster replicate a registry of application instances. Registered applications use the Eureka client to fetch connection and status information about other registered applications, and use this information to connect to instances of other applications.

A client application (here "Producer") instance registers with a *Eureka server* and provides metadata, including host, port, and instance ID. The Producer instance then sends a "heartbeat" message to the Eureka server every 30 seconds. If an application instance fails to send several consecutive heartbeats, the Eureka server removes that instance from its registry.



Each client application (here Consumer) instance also uses the *Eureka client* to fetch the registry from the Eureka server once every 30 seconds (by default). The Consumer instance caches the registry and uses this cache to locate instances of other applications. The Eureka client uses a round-robin load balancing strategy to select a specific instance for each request.

Spring Cloud Eureka Server

Spring Cloud Netflix Eureka provides an embeddable Eureka server, which can be enabled in a Spring Boot application using the `@EnableEurekaServer` annotation. When the Spring Cloud Eureka Server starter (`spring-cloud-starter-eureka-server`) dependency is included, the application becomes a Spring Cloud Netflix Eureka server.

In a Spring Cloud Netflix Eureka server application, a Eureka server configuration can be set using the `eureka.instance` properties in `application.yml` (or `application.properties`).

```
eureka:
  instance:
    hostname: localhost
    nonSecurePort: 80
```

Spring Cloud Eureka Client

The Spring Cloud Commons module includes the `DiscoveryClient` abstraction and `@EnableDiscoveryClient` annotation for applications to use in discovering services, and Spring Cloud Netflix Eureka provides a `DiscoveryClient` implementation for Eureka. When the Spring Cloud Eureka starter (`spring-cloud-starter-eureka`) dependency is included, the application becomes a client of a Spring Cloud Netflix Eureka server.

In a Spring Cloud Netflix Eureka client application, Eureka client configuration can be set using the `eureka.client` properties in `application.yml` (or `application.properties`).

```
eureka:
  client:
    register-with-eureka: true
    leaseRenewalIntervalInSeconds: 10
```

Spring Cloud Services: Service Registry

Spring Cloud Services packages the Spring Cloud Netflix Eureka server application, provisioning one or more instances per Service Registry service instance: one by default, more if the service instance has been created to run in High Availability (HA) mode. This provides operational benefits atop the open-source Spring Cloud library:

- **Application-level registry High Availability (HA).** When creating or updating a Spring Cloud Services Service Registry using the Cloud Foundry Command Line Interface tool (cf CLI), operators can specify a `count` parameter given to the CLI command. The Spring Cloud Services service broker provisions `count` number of Spring Cloud Netflix Eureka server applications for the service instance. These server applications replicate one registry.
- **Zero-downtime blue / green application deployments and upgrades.** The Spring Cloud Services Service Registry permits multiple client applications to register using the same Eureka application name (see [docs page]). Operators can take advantage of this and of Netflix Eureka's instance statuses to deploy updates to applications with zero downtime, using (for example) a blue / green deployment strategy.
- **Peer replication across PCFs / datacenters.** When creating or updating a Spring Cloud Services Service Registry using the cf CLI, operators can specify URIs of other Service Registry service instances in a `peers` parameter given to the CLI command, and the Service Registry service instance replicates its registry with the other service instances. This works for service instances across Pivotal Cloud Foundry spaces, orgs, or deployments, and can be used to provide another level of high availability.

Managing Service Instances

Page last updated:

Creating an Instance

You can create a Service Registry service instance using either the Cloud Foundry Command Line Interface tool (cf CLI) or Pivotal Cloud Foundry® Apps Manager.

Note: Service Registry service instances can run in a high-availability mode, with an arbitrary number of replicated nodes, and can replicate service registrations with peers in other PCF spaces, organizations, or deployments. These behaviors can only be configured through use of the cf CLI. (After creating a service instance using Apps Manager, you can use the cf CLI to configure the instance. See the Updating an Instance section.)

Using the cf CLI

Target the correct org and space:

```
$ cf target -o myorg -s development

API endpoint:   https://api.cf.wise.com (API version: 2.43.0)
User:           user
Org:            myorg
Space:          development
```

You can view plan details for the Config Server product by running `cf marketplace -s`.

```
$ cf marketplace
Getting services from marketplace in org myorg / space development as user...
OK

service                      plans          description
p-circuit-breaker-dashboard  standard       Circuit Breaker Dashboard for Spring Cloud Applications
p-config-server              standard       Config Server for Spring Cloud Applications
p-service-registry           standard       Service Registry for Spring Cloud Applications

TIP:  Use 'cf marketplace -s SERVICE' to view descriptions of individual plans of a given service.

$ cf marketplace -s p-service-registry
Getting service plan information for service p-service-registry as user...
OK

service plan   description     free or paid
standard       Standard Plan   free
```

Run `cf create-service`, specifying the service, plan name, and instance name. Optionally, you may add the `-c` flag and provide a JSON object that specifies configuration parameters.

General parameters used to configure the Service Registry are listed below.

Parameter	Function	Example
`count`	The number of nodes to provision: 1 by default, more for running in high-availability mode	`'{"count": 3}'`

A Service Registry service instance can also be configured to replicate its registry with peer Service Registry service instances in other PCF deployments, organizations, or spaces. For more information on the configuration parameters used to enable peer replication, see the Enabling Peer Replication topic.

To create an instance with the default of a single node, run:

```
$ cf create-service p-service-registry standard service-registry
Creating service instance service-registry in org myorg / space development as user...
OK

Create in progress. Use 'cf services' or 'cf service service-registry' to check operation status.
```

To create an instance, specifying that three nodes should be provisioned, run:

```
$ cf create-service p-service-registry standard service-registry -c '{"count": 3}'
Creating service instance service-registry in org myorg / space development as user...
OK

Create in progress. Use 'cf services' or 'cf service service-registry' to check operation status.
```

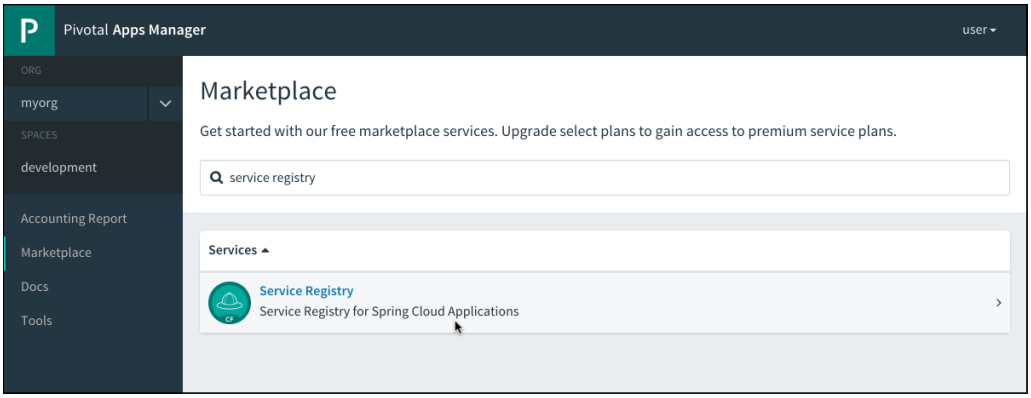
As the command output suggests, you can use the `cf services` or `cf service` commands to check the status of the service instance. When the Service Registry instance is ready to be used, the `cf service` command will give a status of `create succeeded`:

```
$ cf service service-registry

Service instance: service-registry
Service: p-service-registry
Bound apps:
Tags:
Plan: standard
Description: Service Registry for Spring Cloud Applications
Documentation url: http://docs.pivotal.io/spring-cloud-services/
Dashboard: https://spring-cloud-broker.apps.wise.com/dashboard/p-service-registry/50f247f4-fcb0-43c9-863c-94e21be2051c

Last Operation
Status: create succeeded
Message:
Started: 2016-06-27T23:14:44Z
Updated:
```

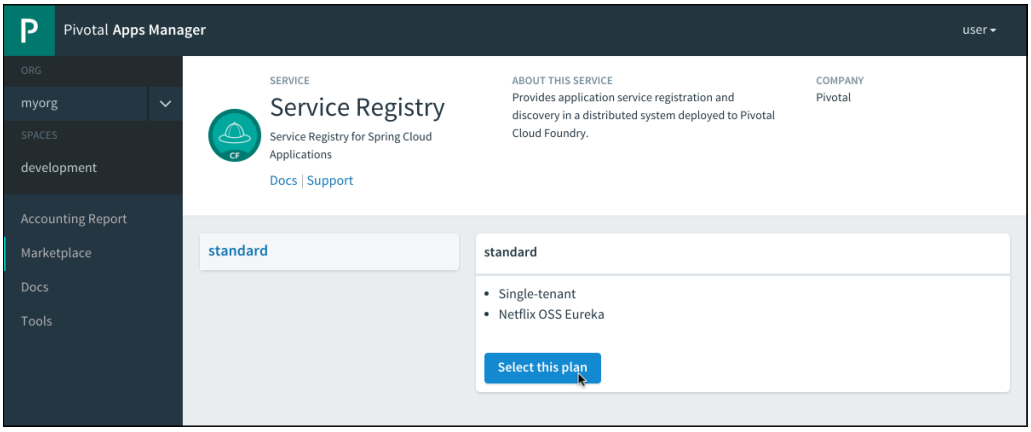
Note: You may notice a discrepancy between the status given for a service instance by the cf CLI (e.g., by the `cf service` command) versus that shown on the Service Instances dashboard. The dashboard is updated frequently (close to real-time); the status retrieved by the cf CLI is not updated as frequently and may take time to match the dashboard.

Using Apps Manager

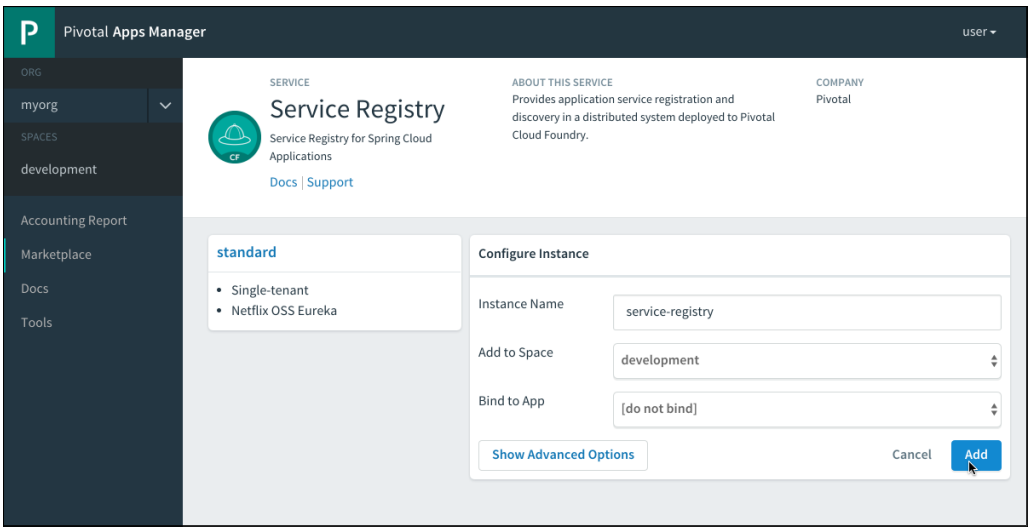
Log into Apps Manager as a Space Developer. In the Marketplace, select **Service Registry**.



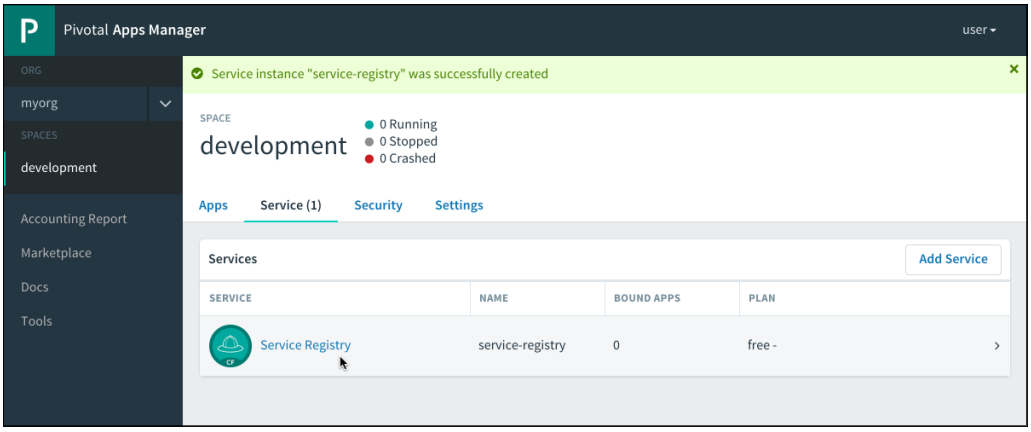
Select the desired plan for the new service instance.



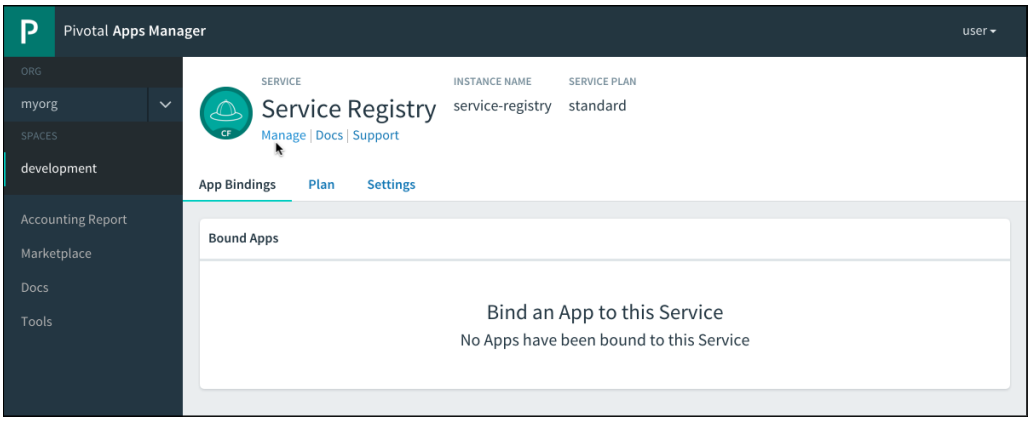
Provide a name for the service instance (for example, “service-registry”). Click the **Add** button.



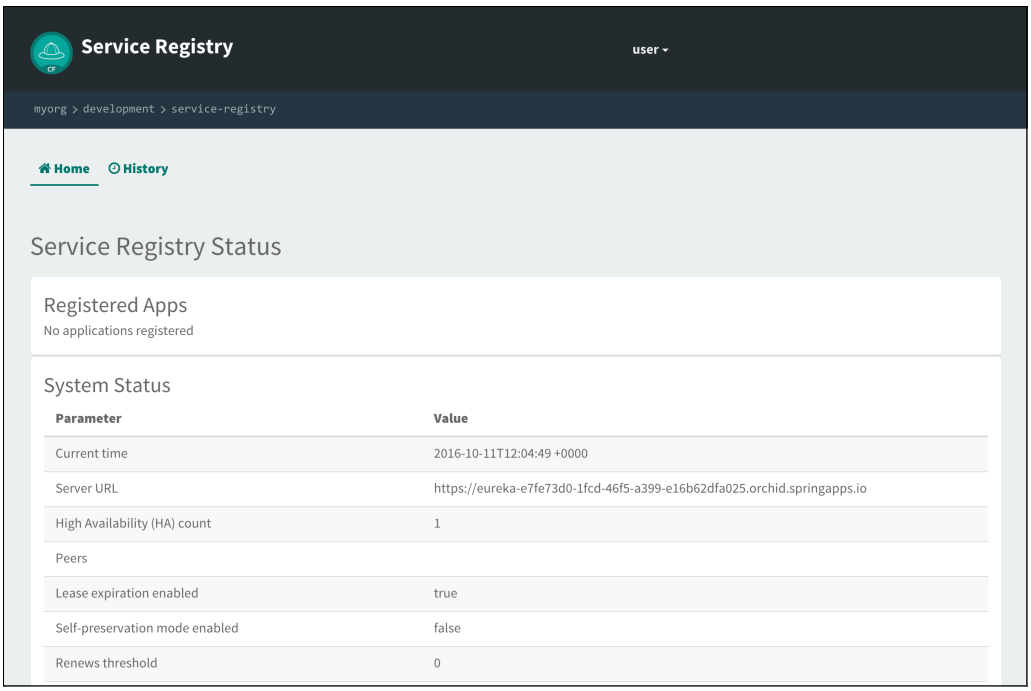
In the **Services** list, click the listing for the new service instance.



Click the **Manage** link.



It may take a few minutes to provision the service instance; while it is being provisioned, you will see a “The service instance is initializing” message. When the instance is ready, its dashboard will load automatically.



For information about registering an application with a Service Registry service instance, see the Writing Client Applications topic.

Updating an Instance

You can update settings on a Service Registry service instance using the Cloud Foundry Command Line Interface tool (cf CLI). The `cf update-service` command can be given a `-c` flag with a JSON object containing parameters used to configure the service instance.

To update a Service Registry service instance's settings, target the org and space of the service instance:

```
$ cf target -o myorg -s development

API endpoint:   https://api.cf.wise.com (API version: 2.43.0)
User:           user
Org:            myorg
Space:          development
```

Then run `cf update-service SERVICE_NAME -c '{ "PARAMETER": "VALUE" }'`, where `SERVICE_NAME` is the name of the service instance, `PARAMETER` is a supported parameter, and `VALUE` is the value for the parameter. For information about supported parameters, see the next section.

Configuration Parameters

General parameters accepted for the Service Registry are listed below.

Parameter	Function	Example
`count`	The number of nodes to provision	`'{"count": 3}'`
`upgrade`	Whether to upgrade the instance	`'{"upgrade": true}'`
`force`	When `upgrade` is set to `true`, whether to force an upgrade of the instance, even if the instance is already at the latest available service version	`'{"force": true}'`

A Service Registry service instance can also be configured to replicate its registry with peer Service Registry service instances in other PCF deployments, organizations, or spaces. For more information on the configuration parameters used to enable peer replication, see the Enabling Peer Replication topic.

To update a service instance and set the count of nodes for running in high-availability mode, run:

```
$ cf update-service service-registry -c '{"count": 3}'
Updating service instance service-registry as user...
OK

Update in progress. Use 'cf services' or 'cf service service-registry' to check operation status.
```

As the command output suggests, you can use the `cf services` or `cf service` commands to check the status of the service instance. When the update is complete, the `cf service` command will give a status of `update succeeded`:

```
$ cf service service-registry

Service instance: service-registry
Service: p-service-registry
Bound apps:
Tags:
Plan: standard
Description: Service Registry for Spring Cloud Applications
Documentation url: http://docs.pivotal.io/spring-cloud-services/
Dashboard: https://spring-cloud-broker.apps.wise.com/dashboard/p-57ce88c1-6bd9-4224-b855-b4600e8e0f39

Last Operation
Status: update succeeded
Message:
Started: 2016-06-28T14:09:19Z
Updated: 2016-06-28T14:12:25Z
```

The service instance is now updated and ready to be used. For information about registering an application with a Service Registry service instance or calling an application which has been registered with a Service Registry service instance, see the Writing Client Applications topic.

Enabling Peer Replication

Page last updated:

You can configure a Service Registry service instance to replicate service registrations with a peer Service Registry service instance. This functionality supports two models:

- **Peer replication across separate Pivotal Cloud Foundry (PCF) deployments:** you can configure peer replication to allow access to services registered with a Service Registry service instance in a PCF deployment located in a separate datacenter.
- **Peer replication across PCF organizations and spaces:** you can configure peer replication to allow access to services registered with a Service Registry service instance in another organization or space within the same PCF deployment.

For information about the configuration parameters used to enable peer replication for a Service Registry service instance, see the Configuration Parameters section.

Configuration Parameters

To enable peer replication for a Service Registry service instance, you must specify the peer Service Registry instance's URI using the `peers` JSON array, which contains an object for each Service Registry peer. You can find a Service Registry service instance's URI on its dashboard (see the Using the Dashboard topic).

A Service Registry peer can be expressed as shown in the following JSON:

```
{
 "peers": [
  { "uri": "https://eureka-e280160b-d3e3-41ad-93a6-479f9b298ca6.wise2.com" }
 ]
}
```

Peer URIs must use the HTTPS URI scheme (as `https://`) and must follow the format `https://eureka-GUID.APPLICATION_DOMAIN.TLD`, where `GUID` is the GUID assigned to a Service Registry service instance and `APPLICATION_DOMAIN.TLD` is the application domain of the PCF deployment where that Service Registry service instance is running.

Spring Cloud Services will by default validate the SSL certificate on each peer. You can disable this validation for a given peer by setting the `skipSslValidation` parameter to `true` for that peer, as shown in the following JSON:

```
{
 "peers": [
  { "uri": "https://eureka-41562ac7-b6a6-4dc2-8a34-c1f94e82c83d.wise2.com",
    "skipSslValidation": true }
 ]
}
```

Note: If you disable certificate validation for a Service Registry service instance's peer, you may need to set the `TRUST_CERTS` environment variable on applications bound to that Service Registry service instance or to the peer. See the Service Registry Peers with Self-Signed Certificates section of the Writing Client Applications topic for more information.

The parameters used to configure a peer for the Service Registry are listed below.

Parameter	Function
`peers[i].uri`	The URI of the Service Registry peer
`peers[i].skipSslValidation`	Whether to skip SSL validation for the Service Registry peer. Valid values are `true` and `false` (default: `false`)

To create or update a Service Registry service instance to replicate a registry with a peer service instance, allowing for validation of the peer's SSL certificate, run one of the following commands:

```
$ cf create-service p-service-registry standard service-registry -c '{ "peers": [ {"uri": "https://eureka-e280160b-d3e3-41ad-93a6-479f9b298ca6.wise2.com"} ] }'

$ cf update-service service-registry -c '{ "peers": [ {"uri": "https://eureka-e280160b-d3e3-41ad-93a6-479f9b298ca6.wise2.com"} ] }'
```

To create or update a Service Registry service instance to replicate a registry with a peer service instance, skipping validation of the peer's SSL certificate, run one of the following commands:

```
$ cf create-service p-service-registry standard service-registry -c '{ "peers": [ {"uri": "https://eureka-e280160b-d3e3-41ad-93a6-479f9b298ca6.wise2.com", "skipSslValidation": true} ] }'

$ cf update-service service-registry -c '{ "peers": [ {"uri": "https://eureka-e280160b-d3e3-41ad-93a6-479f9b298ca6.wise2.com", "skipSslValidation": true} ] }'
```

For an example of configuring peer replication across two Service Registry service instances, see the Setting Up Peer Service Instances section.

Important: If you provide a peer URI which does not correspond to an available Service Registry service instance (e.g. if there is a typo in the URI) or do not disable SSL certificate validation for a peer whose certificate cannot be verified, the `cf create-service` or `cf update-service` command may run successfully and the service instance status may be set to `create succeeded` or `update succeeded`, but the peer will not be used by the Service Registry service instance. In such a case, an error will appear on the Service Registry dashboard. See the Error Conditions section for more information.

Configuration Validation

A peer service instance URI is expected to meet the following criteria:

- Use the HTTPS URI scheme (begin with `https://`). You cannot create or update a Service Registry service instance with a peer URI which does not use the HTTPS URI scheme.
- Follow the pattern `https://eureka-GUID.APPLICATION_DOMAIN.TLD`, where `GUID` is the GUID belonging to the peer service instance and `APPLICATION_DOMAIN.TLD` is the application domain of the PCF deployment on which that service instance is hosted. You cannot create or update a Service Registry service instance with a peer URI which does not follow this pattern.
- Correspond to an available Service Registry service instance. If you create or update a Service Registry service instance with a peer URI which does not correspond to another available service instance, you will see an error message on the Service Registry dashboard.

A peer service instance is expected to meet the following criteria:

- Have the same High Availability (HA) node count as its peer service instance. If you create or update a Service Registry service instance with a peer which has a different HA node count, you will see an error message on the Service Registry dashboard. See the Configuration Parameters section of the Managing Service Instances topic for more information about setting the HA node count for a Service Registry service instance.
- Have a peer list equivalent to its peer service instance's peer list. If you create or update a Service Registry service instance with a peer that does has not have an equivalent peer list, you will see a warning message on the Service Registry dashboard.
- Be at the same Service Registry version as its peer service instance. If you create or update a Service Registry service instance with a peer which is at a different Service Registry version, you will see a warning message on the Service Registry dashboard.

Setting Up Peer Service Instances

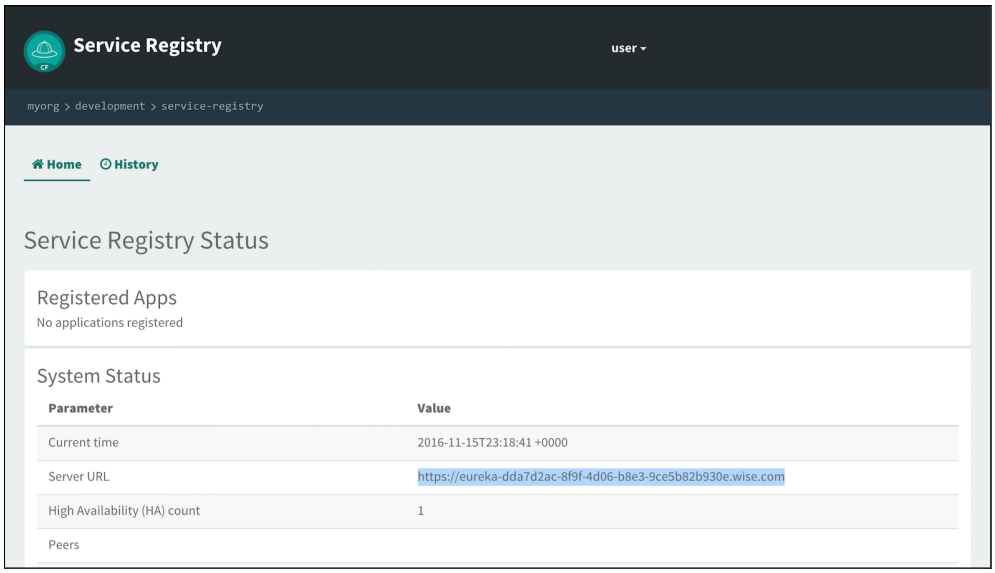
If you wish to have two Service Registry service instances replicate a registry, you must configure each to have the other as a peer. If one service instance Service Registry A has a Service Registry B configured as a peer and B has no peers configured, A will share service registrations with B but B will not share registrations with A.

To configure two-way peer replication for two Service Registry service instances, follow the steps below.

1. Create a Service Registry service instance A, without peers.

```
$ cf create-service p-service-registry standard service-registry
```

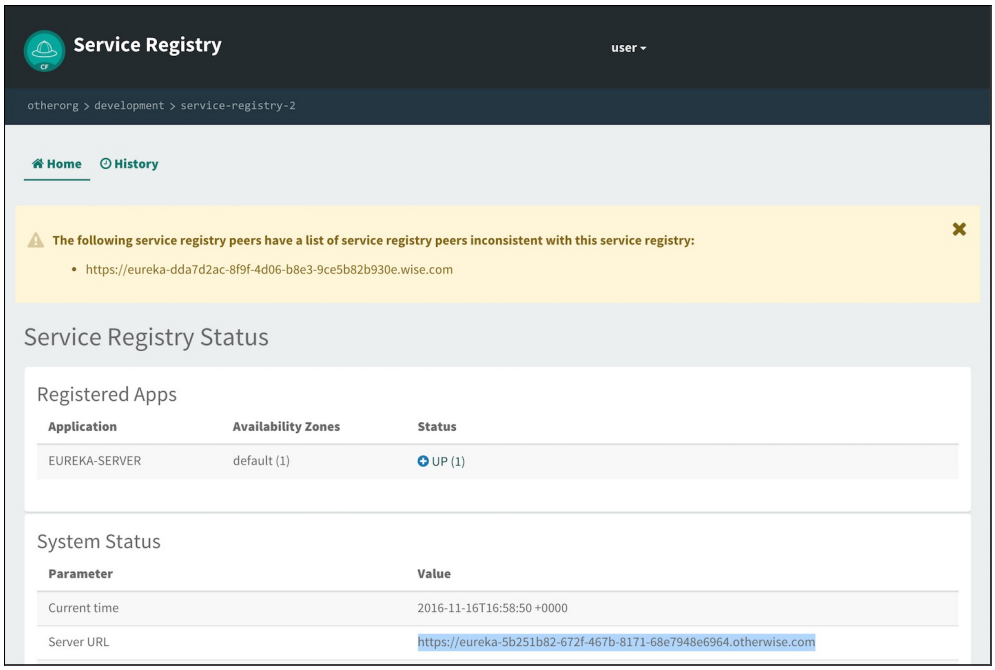
2. Visit A's dashboard and copy its server URL.



3. In the other PCF deployment, organization, or space, create a Service Registry service instance B, with A as a peer.

```
$ cf create-service p-service-registry-2 standard service-registry -c '{ "peers": [ { "uri": "https://eureka-dda7d2ac-8f9f-4d06-b8e3-9ce5b82b930e.wise.com" } ] }'
```

4. Visit B's dashboard and copy its server URL. You will see a warning message on B's dashboard, because B currently has A as a peer and A currently has no peers.



5. Update A, providing B as a peer.

```
$ cf update-service service-registry -c '{ "peers": [ { "uri": "https://eureka-5b251b82-672f-467b-8171-68e7948e6964.otherwise.com" } ] }'
```

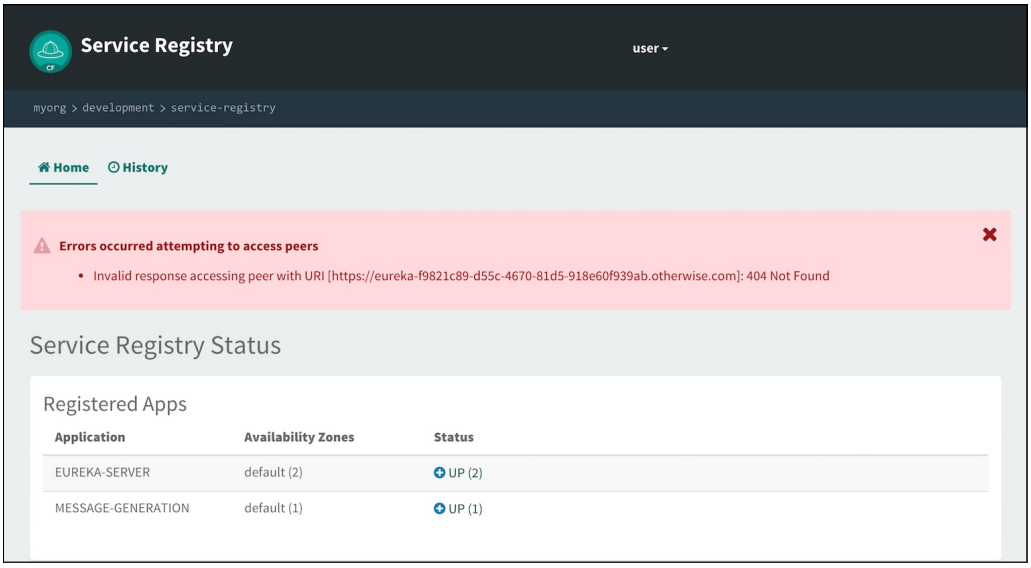
A and B are now configured to replicate service registrations with each other.

Error Conditions

The Service Registry dashboard may show error or warning messages depending on the configuration and status of the service instance's peer configuration. See the below sections for more information.

Peer Service Instance Not Found (404)

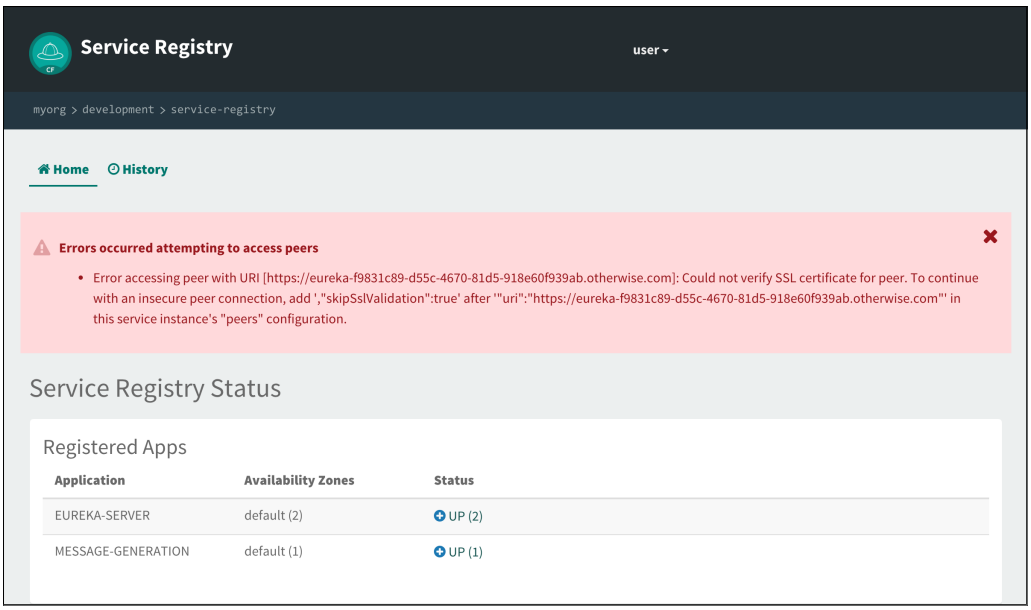
If you have configured a Service Registry service instance with a peer service instance URI that does not correspond to an available Service Registry service instance, you may see the following error message on the dashboard:



Double-check the URI of the peer listed in the error message. This can occur (e.g.) when there is a mistyped URI or when the service instance corresponding to the URI has been deleted.

Peer Service Instance SSL Certificate Not Verified

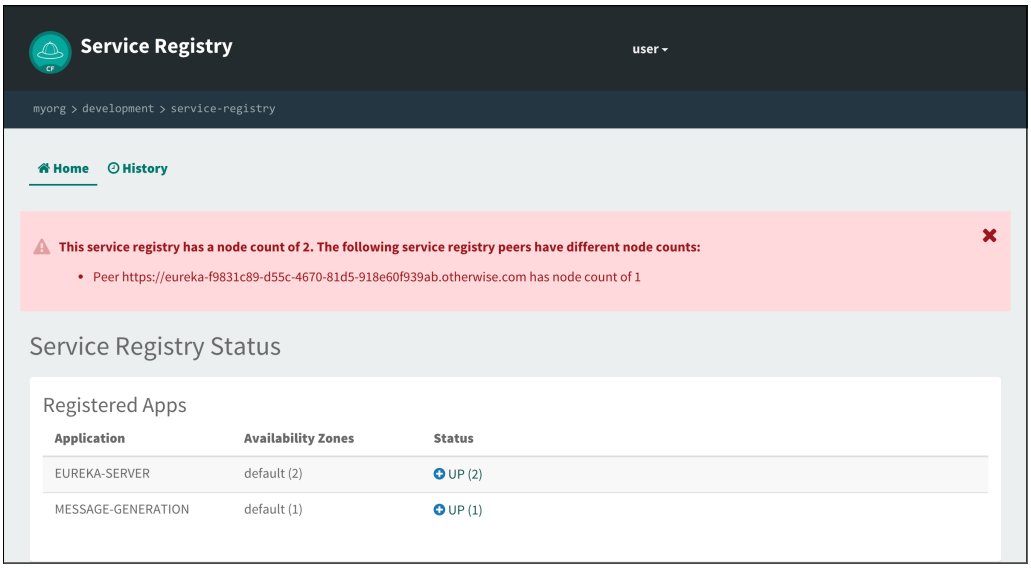
If you have configured a Service Registry service instance with a peer service instance that uses a self-signed SSL certificate and have not disabled certificate validation for the peer service instance, you may see the following error message on the dashboard:



To enable the Service Registry service instance to replicate a registry with this peer service instance without a secure connection, you can update the configuration for this Service Registry service instance to disable SSL certificate validation for the peer. See the Configuration Parameters section for more information about configuring peer replication without SSL certificate validation.

Peer Service Instance Node Count Difference

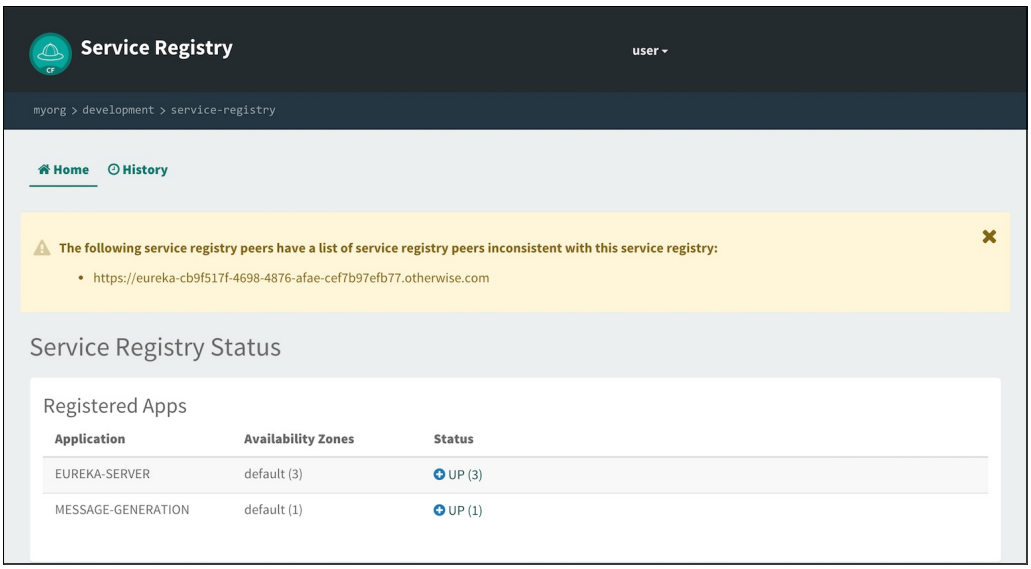
If you have configured a Service Registry service instance with a peer service instance and the two service instances have differing High Availability (HA) node counts, you may see the following error message on the dashboard:



You can update one of the service instances to add or remove nodes so that the two service instances have consistent node counts. See the Configuration Parameters section of the Managing Service Instances topic for more information.

Peer Service Instance Peer List Difference

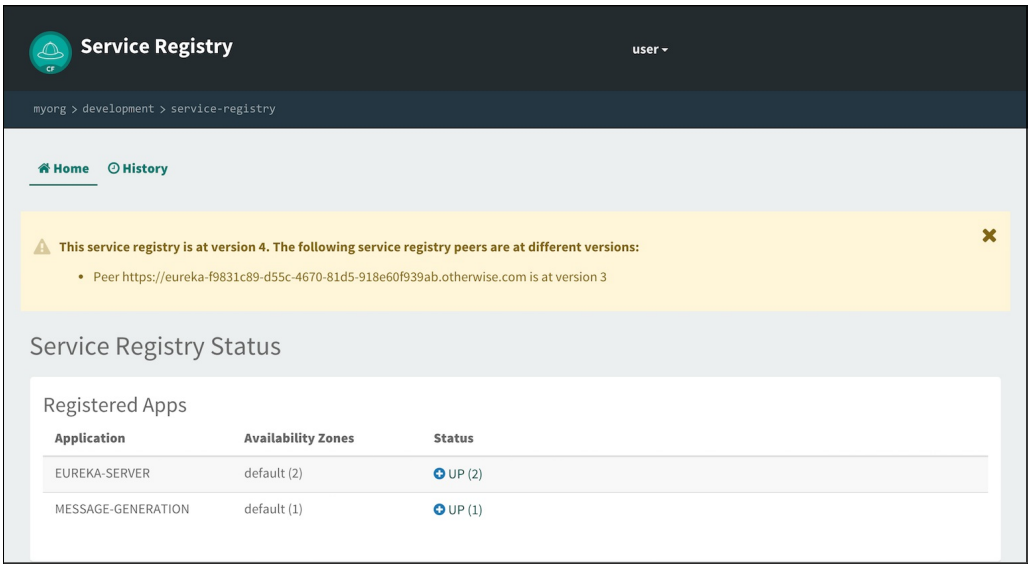
If you have configured a Service Registry service instance with a peer service instance and the two service instances have differing lists of peer service instances, you may see the following message on the dashboard:



Update the configuration for this Service Registry service instance or for the peer instance to give them consistent peer lists.

Peer Service Instance Version Difference

If you have configured a Service Registry service instance with a peer service instance that is at a different version (e.g. that has not been upgraded after an upgrade of the Spring Cloud Services product), you may see the following message on the dashboard:



You can upgrade the older of the service instances so that the peers are at consistent versions. See the Service Instance Upgrades topic for more information.

Writing Client Applications

Page last updated:

Refer to the sample applications in the "greeting" repository to follow along with the code in this topic.

Add Application Dependencies for Service Registry

Important: Ensure that the ordering of the Maven BOM dependencies listed below is preserved in your application's build file. Dependency resolution is affected in both Maven and Gradle by the order in which dependencies are declared.

To work with Spring Cloud Services service instances, your client application must include the `spring-cloud-services-dependencies` and `spring-cloud-dependencies` BOMs. Unless you are using the `spring-boot-starter-parent` or Spring Boot Gradle plugin, you must also specify the `spring-boot-dependencies` BOM as a dependency.

If using Maven, include in `pom.xml`:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.4.1.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>io.pivotal.spring.cloud</groupId>
      <artifactId>spring-cloud-services-dependencies</artifactId>
      <version>1.3.1.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Camden.SR3</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

If not using the `spring-boot-starter-parent`, include in the `<dependencyManagement>` block of `pom.xml`:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>1.4.1.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>

    <!-- ... -->

  </dependencies>
</dependencyManagement>
```

If using Gradle, you will also need to use the Gradle dependency management plugin.

Include in `build.gradle`:

```
buildscript {
  repositories {
    mavenCentral()
  }
  dependencies {
    classpath("io.spring.gradle:dependency-management-plugin:0.6.1.RELEASE")
    classpath("org.springframework.boot:spring-boot-gradle-plugin:1.4.1.RELEASE")
  }
}

apply plugin: "java"
apply plugin: "spring-boot"
apply plugin: "io.spring.dependency-management"

dependencyManagement {
  imports {
    mavenBom "org.springframework.cloud:spring-cloud-dependencies:Camden.SR3"
    mavenBom "io.pivotal.spring.cloud:spring-cloud-services-dependencies:1.3.1.RELEASE"
  }
}

repositories {
  maven {
    url "https://repo.spring.io/plugins-release"
  }
}
```

If not using the Spring Boot Gradle plugin, include in the `dependencyManagement` block of `build.gradle`:

```
dependencyManagement {
  imports {
    mavenBom "org.springframework.boot:spring-boot-dependencies:1.4.1.RELEASE"
  }
}
```

To register your application with a Service Registry service instance or use it to consume a service that is registered with a Service Registry service instance, you must add the `spring-cloud-services-starter-service-registry` dependency.

If using Maven, include in `pom.xml`:

```
<dependencies>
  <dependency>
    <groupId>io.pivotal.spring.cloud</groupId>
    <artifactId>spring-cloud-services-starter-service-registry</artifactId>
  </dependency>
</dependencies>
```

If using Gradle, include in `build.gradle`:

```
dependencies {
  compile("io.pivotal.spring.cloud:spring-cloud-services-starter-service-registry")
}
```

Important: Because of a dependency on Spring Security, the Spring Cloud Services Starters for Service Registry will by default cause all application endpoints to be protected by HTTP Basic authentication. If you wish to disable this, please see Disable HTTP Basic Authentication below.

Add Self-Signed SSL Certificate to JVM Truststore

Spring Cloud® Services uses HTTPS for all client-to-service communication. If your Pivotal Cloud Foundry (PCF) installation is using a self-signed SSL certificate, the certificate will need to be added to the JVM truststore before your application can be registered with a Service Registry service instance or consume a service that is registered with a Service Registry service instance.

Spring Cloud Services can add the certificate for you automatically. For this to work, you must set the `TRUST_CERTS` environment variable on your application to the API endpoint of your Elastic Runtime instance:

```
$ cf set-env message-generation TRUST_CERTS api.cf.wise.com
Setting env variable 'TRUST_CERTS' to 'api.cf.wise.com' for app message-generation in org myorg / space development as user...
OK
TIP: Use 'cf restage message-generation' to ensure your env variable changes take effect
$ cf restage message-generation
```

Note: The `CF_TARGET` environment variable was formerly recommended for configuring Spring Cloud Services to add a certificate to the truststore. `CF_TARGET` is still supported for this purpose, but `TRUST_CERTS` is more flexible and is now recommended instead.

As the output from the `cf set-env` command suggests, restage the application after setting the environment variable.

Service Registry Peers with Self-Signed Certificates

If binding your application to a Service Registry service instance that has one or more peers in another PCF deployment which uses self-signed certificates, you must set the `TRUST_CERTS` environment variable on your application to a hostname on the other PCF deployment in order for your application to communicate with applications bound to those Service Registry peers:

```
$ cf set-env message-generation TRUST_CERTS api.cf.otherwise.com
Setting env variable 'TRUST_CERTS' to 'api.cf.otherwise.com' for app message-generation in org myorg / space development as user...
OK
TIP: Use 'cf restage message-generation' to ensure your env variable changes take effect
```

The `TRUST_CERTS` environment variable can contain multiple hostnames, in case (e.g.) the Service Registry has peers in multiple alternate PCF deployments with self-signed certificates. Hostnames are comma-separated:

```
$ cf set-env message-generation TRUST_CERTS api.cf.otherwise.com,api.verywise.com
Setting env variable 'TRUST_CERTS' to 'api.cf.otherwise.com,api.verywise.com' for app message-generation in org myorg / space development as user...
OK
TIP: Use 'cf restage message-generation' to ensure your env variable changes take effect
```

As the output from the `cf set-env` command suggests, restage the application after setting the environment variable.

```
$ cf restage message-generation
```

Register a Service

To register with a Service Registry service instance, your application must include the `@EnableDiscoveryClient` annotation on a configuration class.

```
@SpringBootApplication
@EnableDiscoveryClient
@RestController
public class MessageGenerationApplication {
  //...
```

The application's Eureka instance name (the name by which it will be registered in Eureka) will be derived from the value of the `spring.application.name` property on the application. If you do not provide a value for this property, the application's Eureka instance name will be derived from its Cloud Foundry application name, as set in `manifest.yml`:

```
---
instances: 1
memory: 1G
applications:
 - name: message-generation
 ...
```

Set the `spring.application.name` property in `application.yml`:

```
spring:
  application:
    name: message-generation
```

Note: If the application name contains characters which are invalid in a hostname, the application will be registered with the Service Registry service instance using the application name with each invalid character replaced by a hyphen (`-`) character (for example, given an application name of "message_generation", the Eureka application name used to register the application with the Service Registry service instance will be `message-generation`). See the Eureka Application Name Configuration section of the Spring Cloud Connectors topic for more information.

Register Using Container-to-Container Networking

To use Cloud Foundry's container networking (see the Understanding Container-to-Container Networking topic in the Pivotal Cloud Foundry documentation) with the application, your `application.yml` must specify a `spring.cloud.services.registrationMethod` of `direct`.

```
spring:
  application:
    name: message-generation
  cloud:
    services:
      registrationMethod: direct
```

Before a client application can use the Service Registry to reach this directly-registered application, you must add a network policy that allows traffic from the client application to this application. See the Consume Using Container-to-Container Networking section for more information.

Consume a Service

Follow the below instructions to consume a service that is registered with a Service Registry service instance.

Discover and Consume a Service Using RestTemplate

A consuming application must include the `@EnableDiscoveryClient` annotation on a configuration class.

```
@SpringBootApplication
@EnableDiscoveryClient
@RestController
public class GreeterApplication {

  @Bean
  @LoadBalanced
  public RestTemplate restTemplate() {
    return new RestTemplate();
  }

  @Autowired
  private RestTemplate rest;
  //...
```

To call a registered service, a consuming application can use a URI with a hostname matching the name with which the service is registered in the Service Registry. This way, the consuming application does not need to know the service application's actual URL; the Registry will take care of finding and routing to the service.

Note: If the name of the registered application contains characters which are invalid in a hostname, that application will be registered with the Service Registry service instance using the application name with each invalid character replaced by a hyphen (`-`) character. For example, given an application name of "message_generation", the Eureka application name used to register the application with the Service Registry service instance will be `message-generation`. See the Eureka Application Name Configuration section of the Spring Cloud Connectors topic for more information.

By default, Service Registry requires HTTPS for access to registered services. If your client application is consuming a service application which has been registered with the Service Registry instance using route registration (see the Register a Service section above), you can use a schemeless URI (as `//message-generation`) to access the service. Spring Cloud Netflix Ribbon will default to using an HTTPS route if one is available and to an HTTP route otherwise. (This behavior requires Spring Cloud Brixton.SR6 or later.)

The Message Generation application is registered with the Service Registry instance as `message-generation`, so in the Greeter application, the `hello()` method on the `GreeterApplication` class uses the base URI `//message-generation` to get a greeting message from Message Generation.

```
@RequestMapping(value = "/hello", method = RequestMethod.GET)
public String hello(@RequestParam(value="salutation",
                    defaultValue="Hello") String salutation,
                @RequestParam(value="name",
                    defaultValue="Bob") String name) {
  URI uri = UriComponentsBuilder.fromUriString("//message-generation/greeting")
    .queryParam("salutation", salutation)
    .queryParam("name", name)
    .build()
    .toUri();

  Greeting greeting = rest.getForObject(uri, Greeting.class);
  return greeting.getMessage();
}
```

Discover and Consume a Service Using Feign

If you wish to use Feign to consume a service that is registered with a Service Registry instance, your application must declare `spring-cloud-starter-feign` as a dependency. In order to have Feign client interfaces automatically configured, it must also use the `@EnableFeignClients` annotation.

Your consuming application must include the `@EnableDiscoveryClient` annotation on a configuration class. In the Greeter application, the `GreeterApplication` class contains a `MessageGenerationClient` interface, which is a Feign client for the Message Generation application.

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
@RestController
public class GreeterApplication {

  @Autowired
  MessageGenerationClient messageGeneration;
  //...
```

To call a registered service, a consuming application can use a URI with a hostname matching the name with which the service is registered in the Service Registry. This way, the consuming application does not need to know the service application's actual URL; the Registry will take care of finding and routing to the service.

Note: If the name of the registered application contains characters which are invalid in a hostname, that application will be registered with the Service Registry service instance using the application name with each invalid character replaced by a hyphen (`-`) character. For example, given an application name of "message_generation", the Eureka application name used to register the application with the Service Registry service instance will be `message-generation`. See the Eureka Application Name Configuration section of the Spring Cloud Connectors topic for more information.

The Message Generation application is registered with the Service Registry instance as `message-generation`, so the `@FeignClient` annotation on the `MessageGenerationClient` interface uses the base URI `http://message-generation`. The interface declares one method, `greeting()`, which accesses the Message Generation application's `/greeting` endpoint and sends along optional `name` and `salutation` parameters if they are provided.

```
@FeignClient("http://message-generation")
interface MessageGenerationClient {
  @RequestMapping(value = "/greeting", method = GET)
  Greeting greeting(@RequestParam("name") String name, @RequestParam("salutation") String salutation);
}
```

Consume Using Container-to-Container Networking

To use Cloud Foundry's container networking (see the Understanding Container-to-Container Networking topic in the Pivotal Cloud Foundry documentation) to reach an application registered with the Service Registry, you must add a network policy. You can do this using the Cloud Foundry Command Line Interface (cf CLI).

Note: Container networking support is included in the cf CLI version 6.30.0 and later.

Run the `cf network-policies` command to list current network policies:

```
$ cf network-policies
Listing network policies in org myorg / space dev as user...

source   destination   protocol   ports
```

Use the `cf add-network-policy` command to grant access from the Greeter application to the Message Generation application:

```
$ cf add-network-policy greeter --destination-app message-generation --protocol tcp --port 8080
Adding network policy to app greeter in org myorg / space dev as user...
OK
```

Use `cf network-policies` again to view the new access policy:

```
$ cf network-policies
Listing network policies in org myorg / space dev as user...

source    destination          protocol   ports
greeter   message-generation   tcp        8080
```

The Greeter application can now use container networking to access the Message Generation application via the Service Registry. For more information about configuring container networking, see the Administering Container-to-Container Networking topic in the Pivotal Cloud Foundry documentation.

Disable HTTP Basic Authentication

The Spring Cloud Services Starter for Service Registry has a dependency on Spring Security. Unless your application has other security configuration, this will cause all application endpoints to be protected by HTTP Basic authentication.

If you do not yet want to address application security, you may turn off Basic authentication by setting the `security.basic.enabled` property to `false`. In `application.yml` or `bootstrap.yml`:

```
security:
  basic:
    enabled: false
```

You might make this setting specific to a profile (such as the `dev` profile if you want Basic authentication disabled only for development):

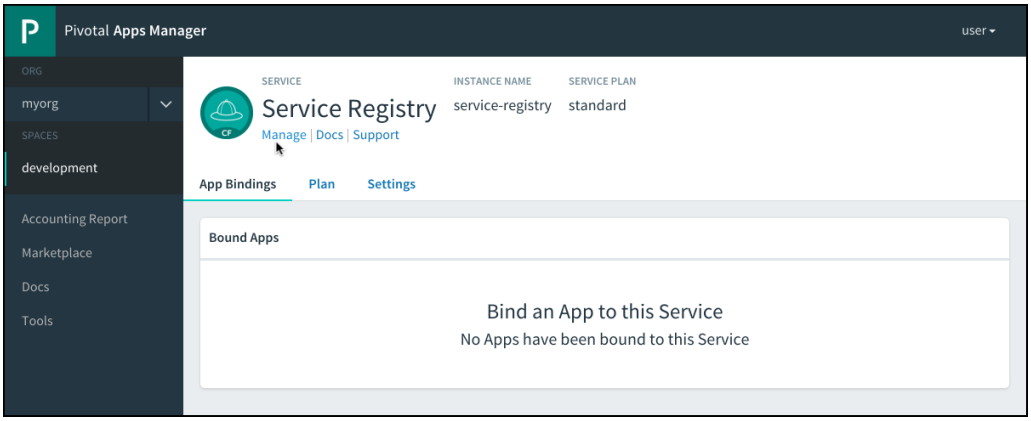
```
---

spring:
  profiles: dev

security:
  basic:
    enabled: false
```

For more information, see "Security" in the Spring Boot Reference Guide.

Using the Dashboard

Page last updated:

To find the dashboard, navigate in Pivotal Cloud Foundry® Apps Manager to the Service Registry service instance's space, click the listing for the service instance, and then click **Manage**.



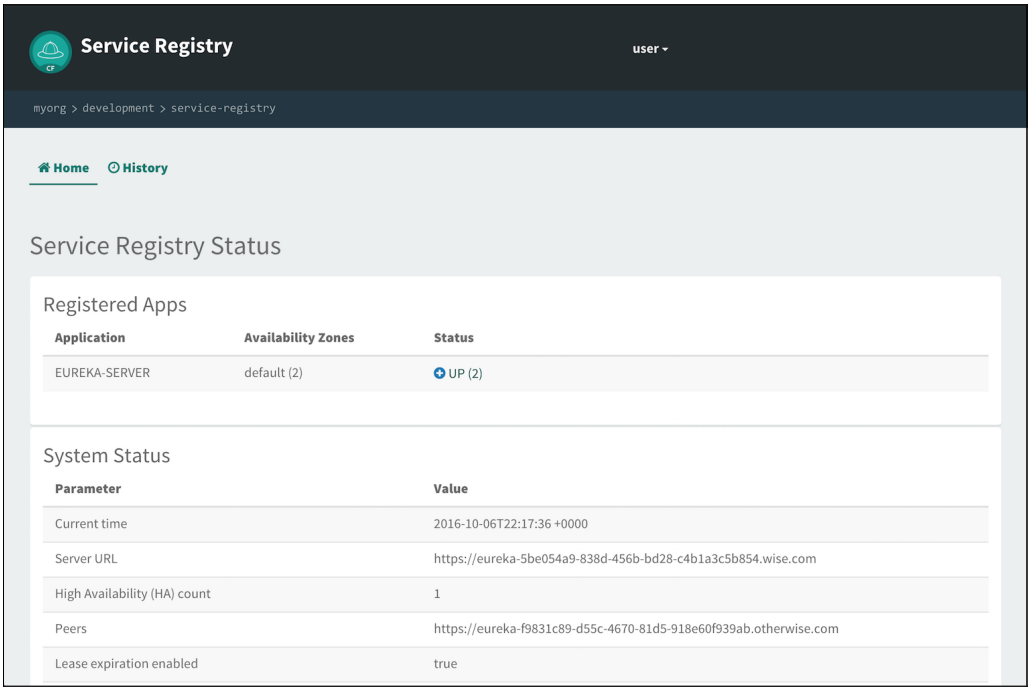
If you are using version 6.8.0 or later of the Cloud Foundry Command Line Interface tool (cf CLI), you can also use `cf service SERVICE_NAME`, where `SERVICE_NAME` is the name of the Service Registry service instance:

```
$ cf service service-registry

Service instance: service-registry
Service: p-service-registry
Bound apps:
Tags:
Plan: standard
Description: Service Registry for Spring Cloud Applications
Documentation url: http://docs.pivotal.io/spring-cloud-services/
Dashboard: https://spring-cloud-broker.wise.com/dashboard/p-service-registry/a9bcaf09-ba0a-4650-bdfc-75784882245e

Last Operation
Status: create succeeded
Message:
Started: 2016-09-19T21:05:32Z
Updated:
```

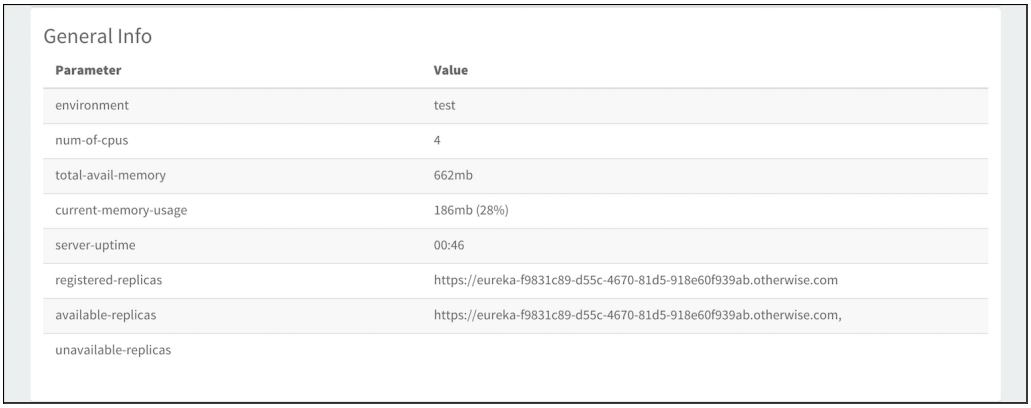
Visit the URL given for "Dashboard".

Dashboard Information

The dashboard shows the URL of the Service Registry service instance, the count of backing applications for the service instance (the High Availability or HA count), and the URLs of any peer service instances, as well as the applications currently registered with the service instance or with any peer service instances.



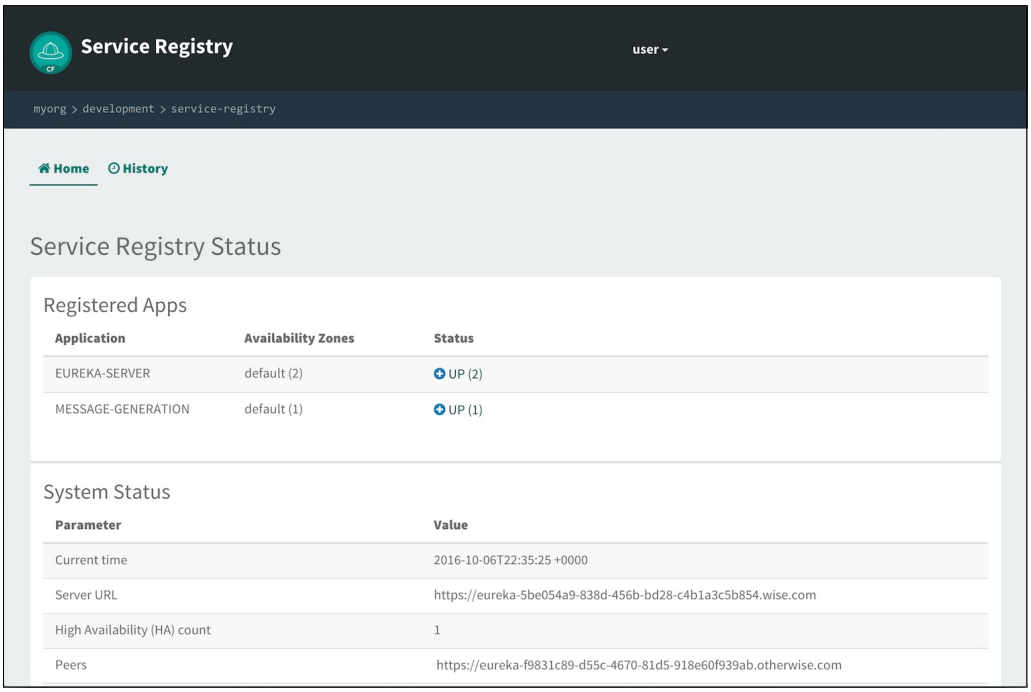
If the Service Registry service instance has been configured to replicate its registry with any peers, peers are listed according to their status in the "replicas" fields under **General Info**. Peers which are registered with this service instance are listed in "registered-replicas" and, if currently in service, will be listed in "available-replicas". Peers which have been configured for this service instance but have not registered will be listed in "unavailable-replicas".



General Info

Parameter	Value
environment	test
num-of-cpus	4
total-avail-memory	662mb
current-memory-usage	186mb (28%)
server-uptime	00:46
registered-replicas	https://eureka-f9831c89-d55c-4670-81d5-918e60f939ab.otherwise.com
available-replicas	https://eureka-f9831c89-d55c-4670-81d5-918e60f939ab.otherwise.com,
unavailable-replicas	

Before you have bound any applications to the Service Registry instance, the Service Registry dashboard displays only `EUREKA-SERVER` (the Service Registry backing application itself) under "Registered Apps". If you bind an application which uses `@EnableDiscoveryClient` to the service instance (see the Writing Client Applications topic), restage the application, and wait for it to register with Eureka, you will see it listed in the dashboard.



Spring Cloud Connectors

Page last updated:

To connect client applications to the Service Registry, Spring Cloud Services uses Spring Cloud Connectors, including the Spring Cloud Cloud Foundry Connector, which discovers services bound to applications running in Cloud Foundry.

Service Detection

The connector inspects Cloud Foundry's `VCAP_SERVICES` environment variable, which stores connection and identification information for service instances that are bound to Cloud Foundry applications, to detect available services. Below is an example of a `VCAP_SERVICES` entry for the Spring Cloud Services Service Registry.

```
{
 "VCAP_SERVICES": {
  "p-service-registry": [
   {
    "credentials": {
     "access_token_uri": "https://p-spring-cloud-services.uaa.cf.wise.com/oauth/token",
     "client_id": "p-service-registry-57bdc399-5b2e-4131-b941-d0a4275c2da4",
     "client_secret": "GAmFDRU4KGnS",
     "uri": "https://eureka-32fb7386-2d57-4054-91b4-9fd4dcbac221.wise.com"
    },
    "label": "p-service-registry",
    "name": "service-registry",
    "plan": "standard",
    "tags": [
     "eureka",
     "discovery",
     "registry",
     "spring-cloud"
    ]
   }
  ]
```

For each service in the `VCAP_SERVICES` variable, the connector considers the following fields:

- `tags`: Attributes or names of backing technologies behind the service.
- `label`: The service offering's name (not to be confused with a service *instance's* name).
- `credentials.uri`: A URI pertaining to the service instance.
- `credentials.uris`: URIs pertaining to the service instance.

Service Registry Detection Criteria

To establish availability of the Service Registry, the Spring Cloud Cloud Foundry Connector compares `VCAP_SERVICES` service entries against the following criteria:

- `tags` including `eureka`

Application Configuration

In a Spring Boot application which is bound to a Service Registry service instance, the connector automatically configures a Spring Cloud Netflix Eureka client configuration bean. The client configuration includes the discovery zone, which is configured using the URL from the service binding; this is equivalent to setting the `eureka.client.serviceUrl.defaultZone` property.

Eureka Application Name Configuration

A client application registers with the Spring Cloud Services Service Registry using an *application name*. This value is used by Netflix Eureka to look up instances of an application and to set Eureka's *virtual hostname* and *secure virtual hostname* for the application. In an application that registers with a Spring Cloud Netflix Eureka server, the Eureka application name is by default the value of the application's `spring.application.name` property. If the value of the application's `spring.application.name` property is used for the Eureka application name and contains one or more characters which are invalid in a hostname, the Spring Cloud Services connector for Service Registry sanitizes that value by using a hyphen (`-`) to replace each invalid character.

You can register a client application with the Service Registry using an application name that differs from `spring.application.name`. To do so, you must set the `eureka.instance.appname` property. If set, this value will be used verbatim for the application's Eureka application name, virtual hostname, and secure virtual hostname; it therefore must contain only characters which are valid in a hostname.

Examples of values provided for the `eureka.instance.appname` property, values provided for the `spring.application.name` property, and the Eureka application names resulting from combinations of these values or from providing a given value for `spring.application.name` but none for `eureka.instance.appname` are listed below.

`eureka.instance.appname`	`spring.application.name`	Resultant application name
vhn	san	vhn
v_hn	san	v_hn
(not provided)	san	san
(not provided)	s_an	s-an
x_y	x_y	x_y

Application Name Property Restrictions

To identify instances of an application, Netflix Eureka uses three `eureka.instance` properties: `appname`, `virtualHostName`, and `secureVirtualHostName`. For consistency with Spring Cloud features such as Spring Cloud Netflix Zuul's automatic routing to registered applications and the load-balanced `RestTemplate` using Spring Cloud Netflix Ribbon, and to secure ownership of Eureka application names, the Spring Cloud Services Service Registry rejects a registration that includes different non-null values for any of these three properties.

If you set only the `spring.application.name` property or the `eureka.instance.appname` property, the Spring Cloud Services connector will set the `eureka.instance.virtualHostName` and `eureka.instance.secureVirtualHostname` properties to the same value. If you explicitly set different values for `eureka.instance.appname`, `eureka.instance.virtualHostName`, and `eureka.instance.secureVirtualHostName`, the Spring Cloud Services connector will prevent the application from starting, and you will see an error message in the application logs:

```
eureka.instance.virtualHostName 'foo' is set to a different value than
eureka.instance.appname 'bar', and is disallowed in Spring Cloud Services.
Try only setting eureka.instance.appname. Please refer to our documentation
and reach out to us if you think you require different values.
```

Instance-Specific Routing in Ribbon

The Spring Cloud Services Connectors include the Cloud Foundry application GUID and instance ID in Ribbon request headers when these values are available. This allows Ribbon to specify a particular instance of an application when making a request. The Gorouter respects such instance-specific routing decisions as made by Ribbon.

Instance-specific routing is performed by Ribbon in the Spring Cloud Services Connectors automatically. For example, if an application has three instances A, B, and C, and A has a Eureka status other than `UP`, Ribbon specifies another instance—either B or C—when making a request to that application, and the Gorouter respects this specific choice by Ribbon even if the Gorouter considers A healthy. The application GUID and instance ID metadata included in Ribbon requests can also be used to write custom routing logic in Ribbon clients within applications using the Spring Cloud Services Services Connectors.

See Also

For more information about Spring Cloud Connectors, see the following:

- Spring Cloud Cloud Foundry Connector documentation
- Spring Cloud Spring Service Connector documentation
- Spring Cloud Connectors documentation
- Spring Cloud Connectors for Spring Cloud Services on Pivotal Cloud Foundry

Additional Resources

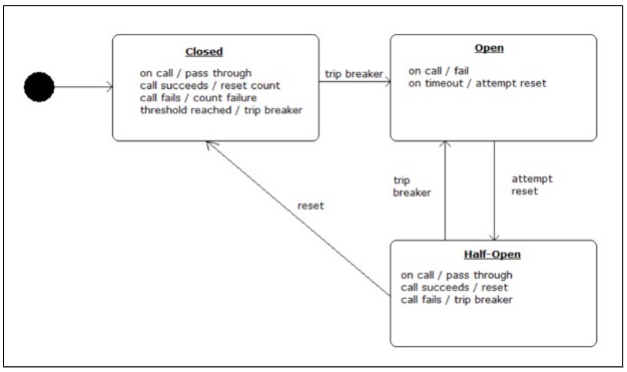
Page last updated:

- Spring Cloud® Services 1.0.0 on Pivotal Cloud Foundry - Service Registry - YouTube (short screencast demonstrating Service Registry for Pivotal Cloud Foundry)
- Home · Netflix/eureka Wiki (documentation for Netflix Eureka, the service registry that underlies Service Registry for Pivotal Cloud Foundry)
- Spring Cloud Netflix (documentation for Spring Cloud Netflix, the open-source project which provides Netflix OSS integrations for Spring applications)
- Microservice Registration and Discovery with Spring Cloud and Netflix's Eureka (discussion and examples of configuring Eureka and Netflix Ribbon using Spring Cloud)
- Client-side service discovery pattern (general description of Service Discovery pattern)
- Service registry pattern (general description of service registry concept)

Circuit Breaker Dashboard

Overview

Circuit Breaker Dashboard provides Spring applications with an implementation of the Circuit Breaker pattern. Cloud-native architectures are typically composed of multiple layers of distributed services. End-user requests may comprise multiple calls to these services, and if a lower-level service fails, the failure can cascade up to the end user and spread to other dependent services. Heavy traffic to a failing service can also make it difficult to repair. Using Circuit Breaker Dashboard, you can prevent failures from cascading and provide fallback behavior until a failing service is restored to normal operation.



When applied to a service, a circuit breaker watches for failing calls to the service. If failures reach a certain threshold, it “opens” the circuit and automatically redirects calls to the specified fallback mechanism. This gives the failing service time to recover.

Circuit Breaker Dashboard is based on Hystrix, Netflix’s latency and fault-tolerance library. For more information about Hystrix and about the Circuit Breaker pattern, see Additional Resources.

Refer to the sample applications in the “traveler” repository to follow along with code in this section.

Managing Service Instances

Page last updated:

Creating an Instance

You can create a Circuit Breaker Dashboard service instance using either the Cloud Foundry Command Line Interface tool (cf CLI) or Pivotal Cloud Foundry (PCF) Apps Manager.

Using the cf CLI

Begin by targeting the correct org and space.

```
$ cf target -o myorg -s development

API endpoint:   https://api.cf.wise.com (API version: 2.43.0)
User:           user
Org:            myorg
Space:          development
```

If desired, view plan details for the Circuit Breaker Dashboard product using `cf marketplace -s`.

```
$ cf marketplace
Getting services from marketplace in org myorg / space development as user...
OK

service                    plans          description
p-circuit-breaker-dashboard   standard       Circuit Breaker Dashboard for Spring Cloud Applications
p-config-server               standard       Config Server for Spring Cloud Applications
p-mysql                    100mb-dev      MySQL service for application development and testing
p-rabbitmq                 standard       RabbitMQ is a robust and scalable high-performance multi-protocol messaging broker.
p-service-registry            standard       Service Registry for Spring Cloud Applications

TIP:  Use 'cf marketplace -s SERVICE' to view descriptions of individual plans of a given service.

$ cf marketplace -s p-circuit-breaker-dashboard
Getting service plan information for service p-circuit-breaker-dashboard as ...
OK

service plan   description     free or paid
standard       Standard Plan   free
```

Run `cf create-service`, specifying the service, plan name, and instance name.

```
$ cf create-service p-circuit-breaker-dashboard standard circuit-breaker-dashboard
Creating service instance circuit-breaker-dashboard in org myorg / space development as admin...
OK

Create in progress. Use 'cf services' or 'cf service circuit-breaker-dashboard' to check operation status.
```

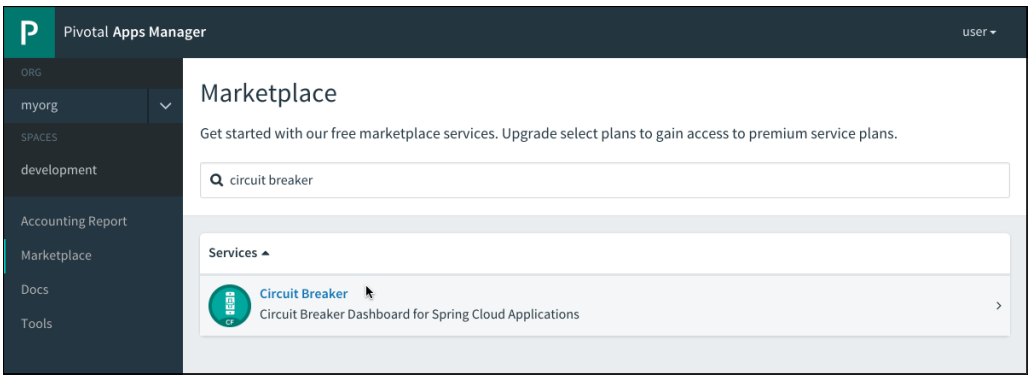
As the command output suggests, you can use the `cf services` or `cf service` commands to check the status of the service instance. When the service instance is ready, the `cf service` command will give a status of `create succeeded`:

```
$ cf service circuit-breaker-dashboard

Service instance: circuit-breaker-dashboard
Service: p-circuit-breaker-dashboard
Bound apps:
Tags:
Plan: standard
Description: Circuit Breaker Dashboard for Spring Cloud® Applications
Documentation url: http://docs.pivotal.io/spring-cloud-services/
Dashboard: https://spring-cloud-broker.apps.wise.com/dashboard/p-circuit-breaker-dashboard/97869d78-5d4e-410b-9c71-bb622ca49f7d

Last Operation
Status: create succeeded
Message:
Started: 2016-06-29T19:13:13Z
Updated: 2016-06-29T19:16:19Z
```

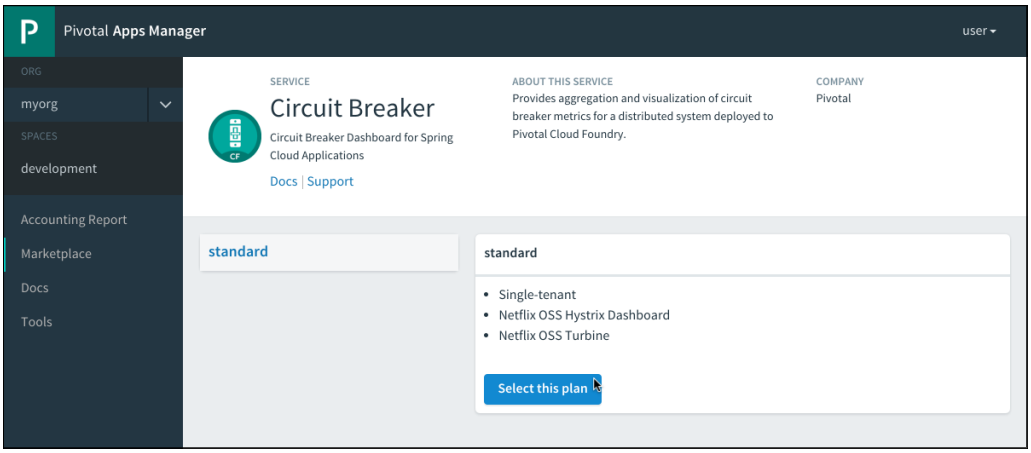
Note: You may notice a discrepancy between the status given for a service instance by the cf CLI (e.g., by the `cf service` command) versus that shown on the Service Instances dashboard. The dashboard is updated frequently (close to real-time); the status retrieved by the cf CLI is not updated as frequently and may take time to match the dashboard.

Using Apps Manager

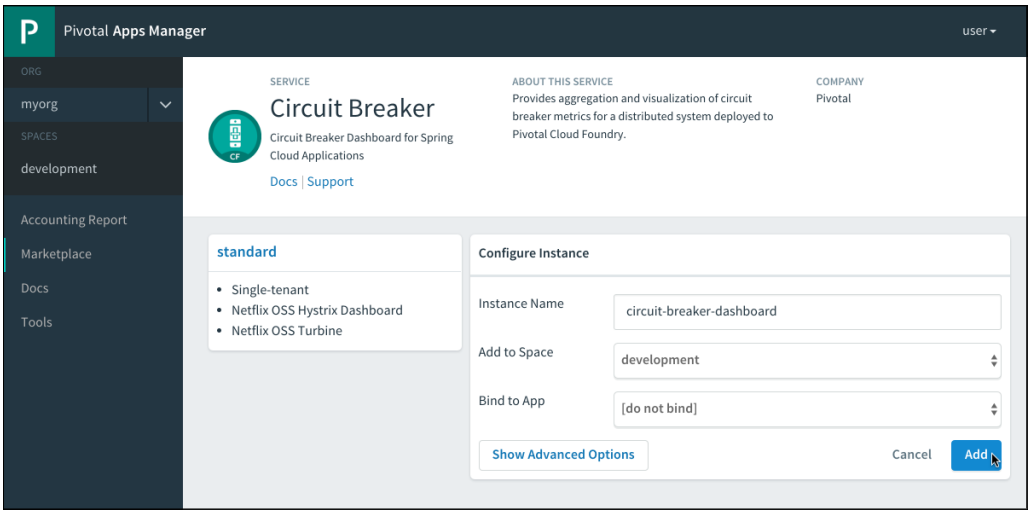
Log into Apps Manager as a Space Developer. In the Marketplace, select **Circuit Breaker**.



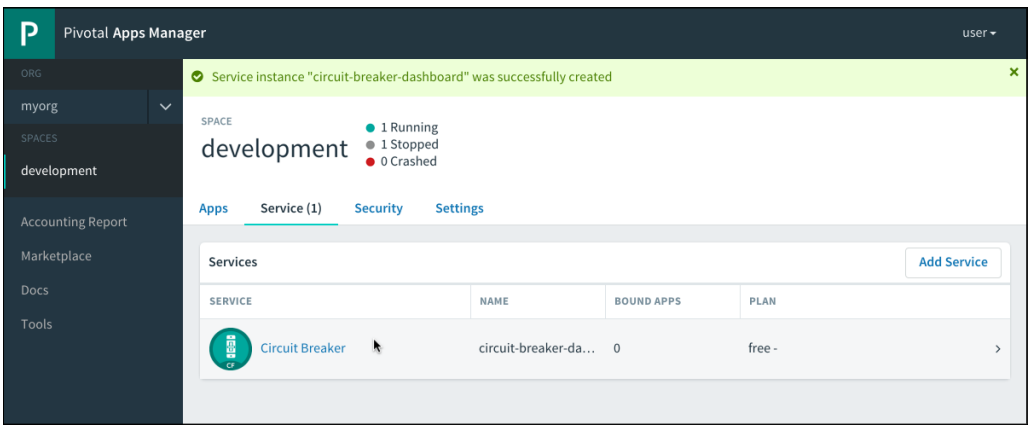
Select the desired plan for the new service instance.



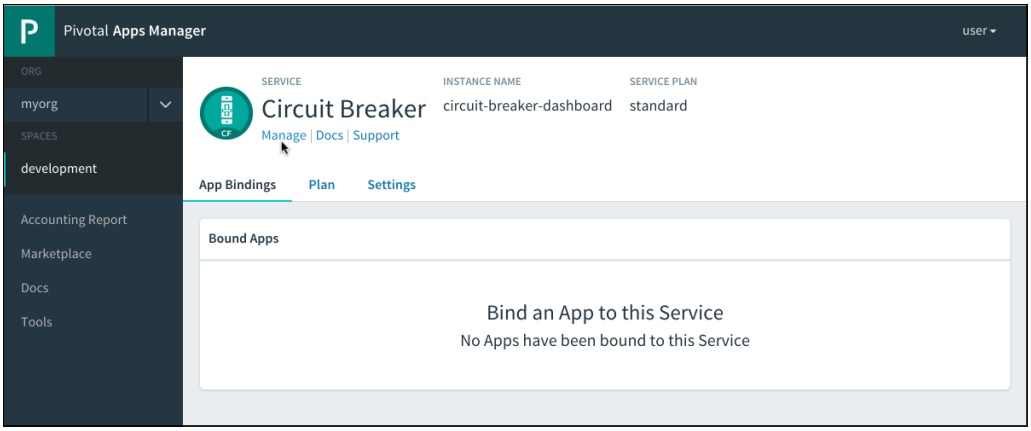
Provide a name for the service instance (for example, “circuit-breaker-dashboard”). Click the **Add** button.



In the **Services** list, click the listing for the new service instance.



Click the **Manage** link.



It may take a few minutes to provision the service instance; while it is being provisioned, you will see a “The service instance is initializing” message. Once the instance is ready, its dashboard will load automatically.

The Circuit Breaker Dashboard instance is now ready to be used. For an example of using a circuit breaker in an application, see the Writing Client Applications topic.

Updating an Instance

You can update settings on a Circuit Breaker Dashboard service instance using the Cloud Foundry Command Line Interface tool (cf CLI). The `cf update-service` command can be given a `-c` flag with a JSON object containing parameters used to configure the service instance.

To update a Circuit Breaker Dashboard service instance’s settings, target the org and space of the service instance:

```
$ cf target -o myorg -s development

API endpoint:   https://api.cf.wise.com (API version: 2.43.0)
User:           user
Org:            myorg
Space:          development
```

Then run `cf update-service SERVICE_NAME -c '{ "PARAMETER": "VALUE" }'`, where `SERVICE_NAME` is the name of the service instance, `PARAMETER` is a supported parameter, and `VALUE` is the value for the parameter. For information about supported parameters, see the next section.

Configuration Parameters

General parameters accepted for the Circuit Breaker Dashboard are listed below.

Parameter	Function	Example
`upgrade`	Whether to upgrade the instance	`'{"upgrade": true}'`
`force`	When `upgrade` is set to `true`, whether to force an upgrade of the instance, even if the instance is already at the latest available service version	`'{"force": true}'`

To update a service instance so that it is upgraded to the latest available service version, run:

```
$ cf update-service circuit-breaker-dashboard -c '{"upgrade": true}'
Updating service instance circuit-breaker-dashboard as user...
OK

Update in progress. Use 'cf services' or 'cf service circuit-breaker-dashboard' to check operation status.
```

As the command output suggests, you can use the `cf services` or `cf service` commands to check the status of the service instance. When the update is complete, the `cf service` command will give a status of `update succeeded`:

```
$ cf service circuit-breaker-dashboard

Service instance: circuit-breaker-dashboard
Service: p-circuit-breaker-dashboard
Bound apps: agency
Tags:
Plan: standard
Description: Circuit Breaker Dashboard for Spring Cloud Applications
Documentation url: http://docs.pivotal.io/spring-cloud-services/
Dashboard: https://spring-cloud-broker.apps.wise.com/dashboard/p-circuit-breaker-dashboard/9397f120-484d-45ef-a7bd-20661e98b643

Last Operation
Status: update succeeded
Message:
Started: 2017-06-16T18:18:14Z
Updated: 2017-06-16T18:20:17Z
```

The service instance is now updated and ready to be used. For an example of using a circuit breaker in an application, see the Writing Client Applications topic.

Writing Client Applications

Page last updated:

Refer to the sample applications in the "traveler" repository to follow along with the code in this topic.

To use a circuit breaker in a Spring application with a Circuit Breaker Dashboard service instance, you must add the dependencies listed in the Client Dependencies topic to your application's build file. Be sure to include the dependencies for Circuit Breaker Dashboard as well.

Use a Circuit Breaker

To work with a Circuit Breaker Dashboard instance, your application must include the `@EnableCircuitBreaker` annotation on a configuration class.

```
import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;
//...

@SpringBootApplication
@EnableDiscoveryClient
@RestController
@EnableCircuitBreaker
public class AgencyApplication {
  //...
```

To apply a circuit breaker to a method, annotate the method with `@HystrixCommand`, giving the annotation the name of a `fallbackMethod`.

```
  @HystrixCommand(fallbackMethod = "getBackupGuide")
  public String getGuide() {
    return restTemplate.getForObject("http://company/available", String.class);
  }
```

The `getGuide()` method uses a `RestTemplate` to obtain a guide name from another application called Company, which is registered with a Service Registry instance. (See the Service Registry documentation, specifically the Writing Client Applications topic.) The method thus relies on the Company application to return a response, and if the Company application fails to do so, calls to `getGuide()` will fail. When the failures exceed the threshold, the breaker on `getGuide()` will open the circuit.

While the circuit is open, the breaker redirects calls to the annotated method, and they instead call the designated `fallbackMethod`. The fallback method must be in the same class and have the same method signature (i.e., have the same return type and accept the same parameters) as the annotated method. In the Agency application, the `getGuide()` method on the `TravelAgent` class falls back to `getBackupGuide()`.

```
  String getBackupGuide() {
    return "None available! Your backup guide is: Cookie";
  }
```

If you wish, you may also annotate fallback methods themselves with `@HystrixCommand` to create a fallback chain.

Use a Circuit Breaker with a Feign Client

You cannot apply `@HystrixCommand` directly to a Feign client interface at this time. Instead, you can call Feign client methods from a service class that is autowired as a Spring bean (either through the `@Service` or `@Component` annotations or by being declared as a `@Bean` in a configuration class) and then annotate the service class methods with `@HystrixCommand`.

In the Feign version of the Agency application, the `AgencyApplication` class is annotated with `@EnableFeignClients`.

```
//...
import org.springframework.cloud.netflix.feign.EnableFeignClients;

@SpringBootApplication
@EnableDiscoveryClient
@RestController
@EnableCircuitBreaker
@EnableFeignClients
public class AgencyApplication {
  // ...
```

The application has a Feign client called `CompanyClient`.

```
package agency;

import org.springframework.stereotype.Component;
import org.springframework.cloud.netflix.feign.FeignClient;
import org.springframework.web.bind.annotation.RequestMapping;

import static org.springframework.web.bind.annotation.RequestMethod.GET;

@FeignClient("https://company")
interface CompanyClient {
  @RequestMapping(value="/available", method = GET)
  String availableGuide();
}
```

The `TravelAgent` class is annotated as a `@Service` class. The `CompanyClient` class is injected through autowiring, and the `getGuide()` method uses the `CompanyClient` to access the Company application. `@HystrixCommand` is applied to the service method:

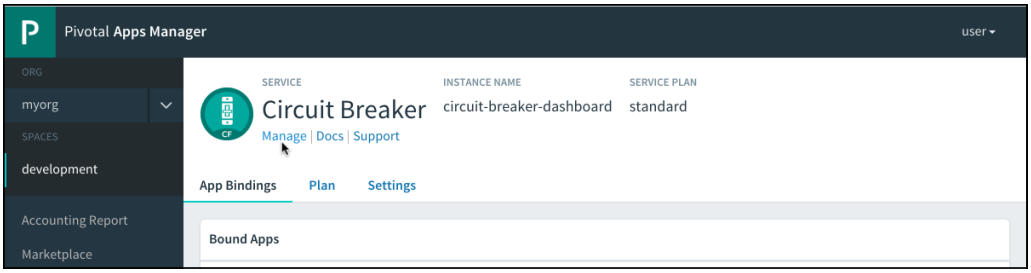
```
package agency;

import org.springframework.stereotype.Service;
import org.springframework.beans.factory.annotation.Autowired;

import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;

@Service
public class TravelAgent {

   @Autowired
   CompanyClient company;

   @HystrixCommand(fallbackMethod = "getBackupGuide")
   public String getGuide() {
      return company.availableGuide();
   }

   String getBackupGuide() {
      return "None available! Your backup guide is: Cookie";
   }

}
```

If the Company application becomes unavailable or if the Agency application cannot access it, calls to `getGuide()` will fail. When successive failures build up to the threshold, Hystrix will open the circuit, and subsequent calls will be redirected to the `getBackupGuide()` method until the Company application is accessible again and the circuit is closed.

Using the Dashboard

Page last updated:

To find the dashboard, navigate in Pivotal Cloud Foundry® Apps Manager to the Circuit Breaker Dashboard service instance's space, click the listing for the service instance, and then click **Manage**.



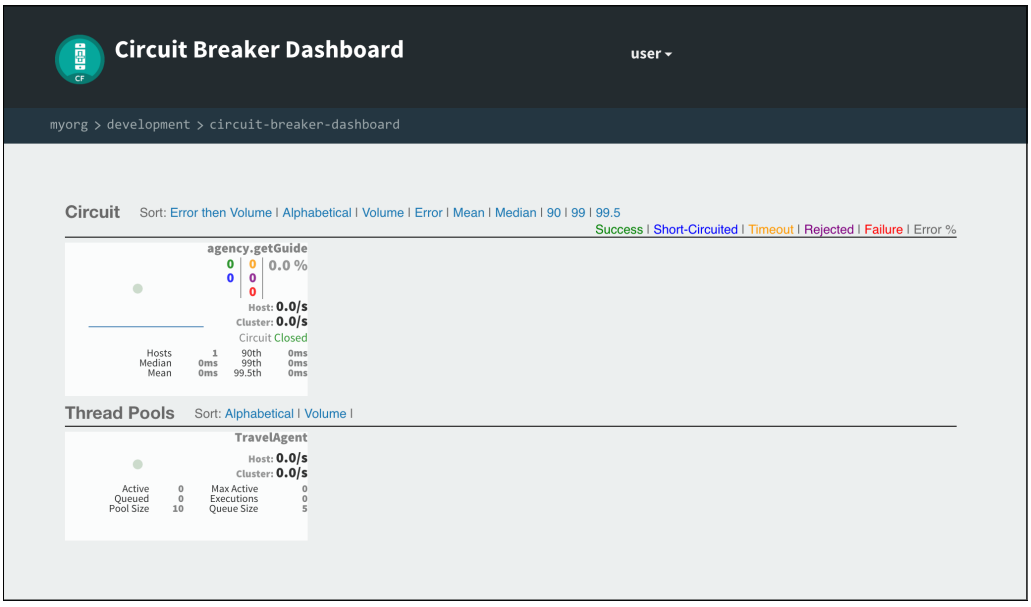
If you are using version 6.8.0 or later of the Cloud Foundry Command Line Interface (cf CLI), you can also use `cf service SERVICE_NAME`, where `SERVICE_NAME` is the name of the Circuit Breaker Dashboard service instance:

```
$ cf service circuit-breaker-dashboard

Service instance: circuit-breaker-dashboard
Service: p-circuit-breaker-dashboard
Bound apps:
Tags:
Plan: standard
Description: Circuit Breaker Dashboard for Spring Cloud® Applications
Documentation url: http://docs.pivotal.io/spring-cloud-services/
Dashboard: https://spring-cloud-broker.apps.wise.com/dashboard/p-circuit-breaker-dashboard/97869d78-5d4e-410b-9c71-bb622ca49f7d

Last Operation
Status: create succeeded
Message:
Started: 2016-06-29T19:13:13Z
Updated: 2016-06-29T19:16:19Z
```

Visit the URL given for "Dashboard".

To see breaker statuses on the dashboard, configure an application as described in the Writing Client Applications topic, using `@HystrixCommand` annotations to apply circuit breakers. Then push the application and bind it to the Circuit Breaker Dashboard service instance. Once bound and restaged, the application will update the dashboard with metrics that describe the health of its monitored service calls.

With the "Agency" example application (see the "traveler" repository) receiving no load, the dashboard displays the following:

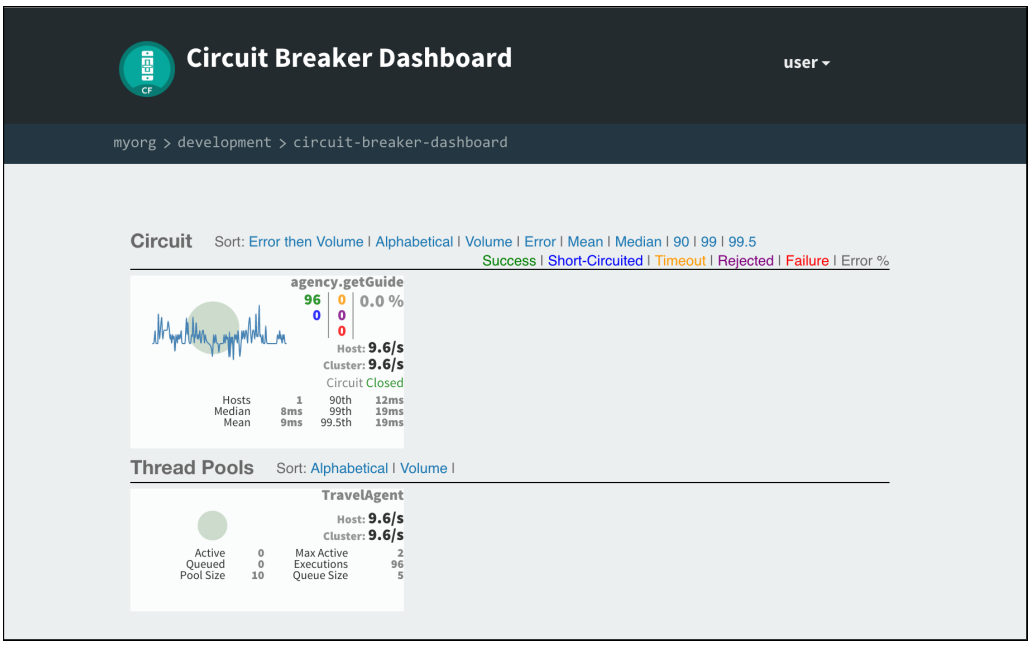


To see the circuit breaker in action, use curl, JMeter, Apache Bench, or similar to simulate load.

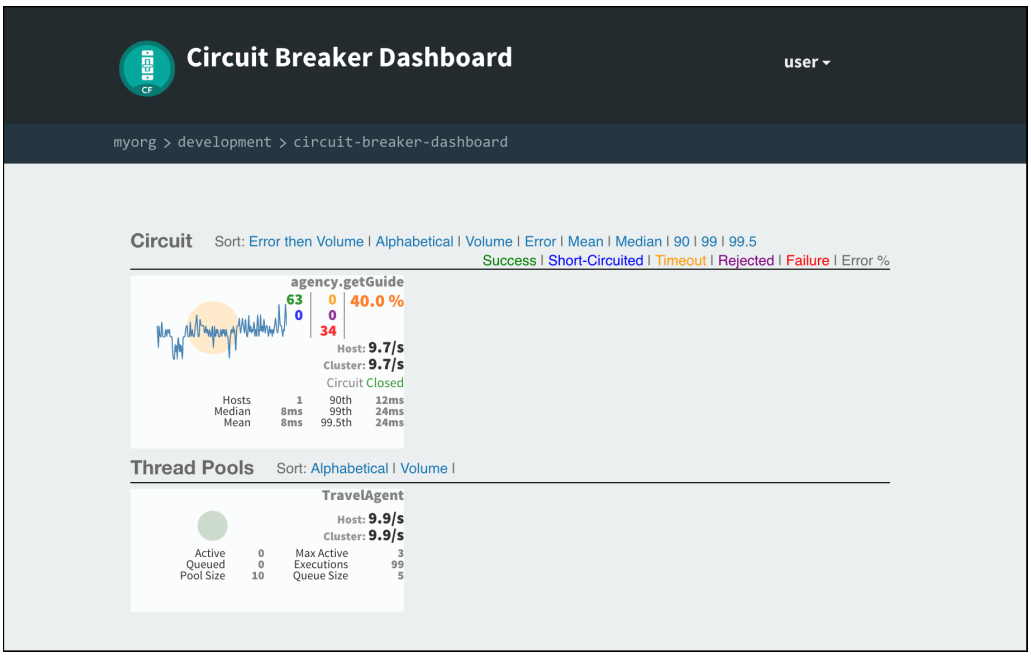
```
$ while true; do curl agency.wise.com; done
```

With the Company application running and available via the Service Registry instance (see the Writing Client Applications topic), the Agency application responds with a guide name, indicating a successful service call. If you stop Company, Agency will respond with a "None available" message, indicating that the call to its `getGuide()` method failed and was redirected to the fallback method.

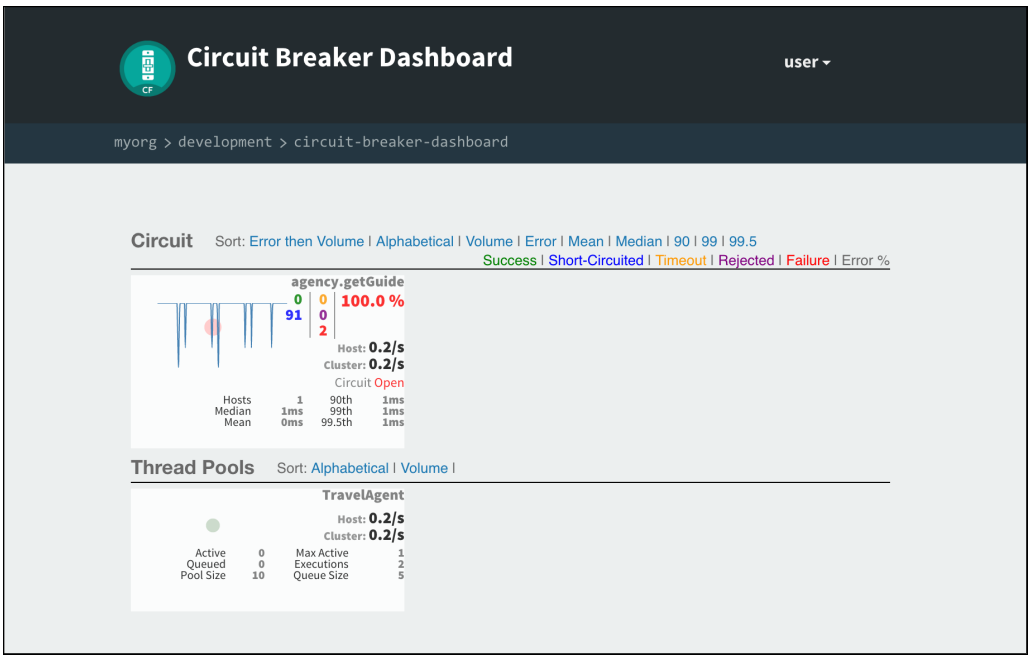
When service calls are succeeding, the circuit is closed, and the dashboard graph shows the rate of calls per second and successful calls per 10 seconds.



When calls begin to fail, the graph shows the rate of failed calls in red.



When failures exceed the configured threshold (the default is 20 failures in 5 seconds), the breaker opens the circuit. The dashboard shows the rate of short-circuited calls—calls which are going straight to the fallback method—in blue. The application is still allowing calls to the failing method at a rate of 1 every 5 seconds, as indicated in red; this is necessary to determine if calls are succeeding again and if the circuit can be closed.



With the circuit breaker in place on its `getGuide()` method, the Agency example application never returns an HTTP status code other than `200` to the requester.

Spring Cloud® Connectors

Page last updated:

To connect client applications to the Circuit Breaker Dashboard, Spring Cloud Services uses Spring Cloud Connectors, including the Spring Cloud Cloud Foundry Connector, which discovers services bound to applications running in Cloud Foundry.

Service Detection

The connector inspects Cloud Foundry's `VCAP_SERVICES` environment variable, which stores connection and identification information for service instances that are bound to Cloud Foundry applications, to detect available services. Below is an example excerpt of a `VCAP_SERVICES` entry for the Spring Cloud Services Circuit Breaker Dashboard (edited for brevity).

```
{
 "VCAP_SERVICES": {
  "p-circuit-breaker-dashboard": [
   {
    "credentials": {
     "dashboard": "https://hystrix-67b8d606-c287-4fb1-9b3f-bd5cb27a29b5.wise.com",
     "stream": "https://turbine-67b8d606-c287-4fb1-9b3f-bd5cb27a29b5.wise.com"
    },
    "label": "p-circuit-breaker-dashboard",
    "name": "circuit-breaker-dashboard",
    "plan": "standard",
    "tags": [
     "circuit-breaker",
     "hystrix-amqp",
     "spring-cloud"
    ]
   }
  ]
```

For each service in the `VCAP_SERVICES` variable, the connector considers the following fields:

- `tags` : Attributes or names of backing technologies behind the service.
- `label` : The service offering's name (not to be confused with a service *instance's* name).
- `credentials.uri` : A URI pertaining to the service instance.
- `credentials.uris` : URIs pertaining to the service instance.

Circuit Breaker Dashboard Detection Criteria

To establish availability of the Circuit Breaker Dashboard, the Spring Cloud Cloud Foundry Connector compares `VCAP_SERVICES` service entries against the following criteria:

- `tags` including `hystrix-amqp`

Metrics Collection with Spring Cloud Stream

A Circuit Breaker Dashboard service instance uses Spring Cloud Stream to collect Netflix Hystrix metrics from a client application. The connector creates a `hystrix` configuration of the Spring Cloud Stream RabbitMQ binder and configures a `hystrixStreamOutput` channel, over which a client application sends Hystrix metrics.

The `defaultCandidate` property on the `hystrix` RabbitMQ binder configuration is set to `false`, so that this binder configuration will not affect any default binder configuration in the client application and you can freely configure Spring Cloud Stream binders (including the RabbitMQ binder) in the client application. For more information, see "Connecting to Multiple Systems" in the Spring Cloud Stream Reference Guide.

See Also

For more information about Spring Cloud Connectors, see the following:

- Spring Cloud Cloud Foundry Connector documentation
- Spring Cloud Spring Service Connector documentation
- Spring Cloud Connectors documentation
- Spring Cloud Connectors for Spring Cloud Services on Pivotal Cloud Foundry

Additional Resources

Page last updated:

- Spring Cloud® Services 1.0.0 on Pivotal Cloud Foundry - Circuit Breaker - YouTube (short screencast demonstrating Circuit Breaker Dashboard)
- Home · Netflix/Hystrix Wiki (documentation for Netflix Hystrix, the library that underlies Circuit Breaker Dashboard)
- Spring Cloud Netflix (documentation for Spring Cloud Netflix, the open-source project which provides Netflix OSS integrations for Spring applications)
- CircuitBreaker (article in Martin Fowler's Bliki about the Circuit Breaker pattern)
- The Netflix Tech Blog: Introducing Hystrix for Resilience Engineering (Netflix Tech introduction of Hystrix)
- The Netflix Tech Blog: Making the Netflix API More Resilient (Netflix Tech description of how Netflix's API used the Circuit Breaker pattern)
- SpringOne2GX 2014 Replay: Spring Boot and Netflix OSS (59:54 in the video begins an introduction of Hystrix and a Spring Cloud demo)