Pivotal

PRODUCT DOCUMENTATION

Spring Cloud Services for PCF Documentation

Version 1.0

Published: 5 Nov 2018

Rev: 01

Spring Cloud Services for Pivotal Cloud Foundry®

Spring Cloud Services for Pivotal Cloud Foundry® (PCF) packages server-side components of Spring Cloud projects, such as Spring Cloud Netflix and Spring Cloud Config, and makes them available as services in the Marketplace. This frees you from having to implement and maintain your own managed services in order to use the included projects. Taking advantage of these battle-tested patterns, and of the libraries that implement them, can now be as simple as including a few dependencies in your application build file and applying the appropriate annotation in your source code.

Services

Spring Cloud Services currently provides the following services.

Service Type	Current Version
Config Server	2
Service Registry	2
Circuit Breaker Dashboard	2

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

Dependencies

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

Service	Supported Versions
MySQL	1.6.1-1.7.x
RabbitMQ	1.4.0-1.5.x

Product Snapshot

Current Spring Cloud Services for PCF Details

- **Version**: 1.0.45
- **Release Date**: 27 June 2018
- **Software component version**: Spring Cloud OSS Angel.SR4
- **Compatible Ops Manager Version(s)**: 1.7.x, 1.6.x
- **Compatible Elastic Runtime Version(s)**: 1.7.x, 1.6.x
- **vSphere support?** Yes
- **AWS support?** Yes

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.

Platform and Product Requirements

Spring Cloud Services is compatible with the Java buildpack at the version shipped with Pivotal Cloud Foundry® Elastic Runtime or later. At minimum, Spring Cloud Services requires version 2.5 or later of the Java buildpack; it is recommended that you use the latest buildpack version possible. 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-v3.0.zip
ruby_buildpack          2          true      false    ruby_buildpack-cached-v1.3.1.zip
nodejs_buildpack        3          true      false    nodejs_buildpack-cached-v1.2.1.zip
go_buildpack            4          true      false    go_buildpack-cached-v1.2.0.zip
```

If the default Java buildpack is older than version 2.5, you can download a newer version from Pivotal Network and update Pivotal Cloud Foundry® by following the instructions in the Adding Buildpacks to Cloud Foundry topic. If you do not delete or disable the older Java buildpack, make sure that the newer Java buildpack is in a lower position so that it will be the default.

Spring Cloud Services also requires the following Pivotal Cloud Foundry® products:

- MySQL
- RabbitMQ

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

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.
```

Self-Signed Certificates and Internal CAs

If your Pivotal Cloud Foundry® installation is using its own self-signed SSL certificate, you must enable the “Trust Self-Signed Certificates” or “Ignore SSL certificate verification” option in the **Networking** section of the Pivotal Elastic Runtime tile **Settings** tab. See Step 4 of the Configuring Elastic Runtime for vSphere and vCloud topic in the Pivotal Cloud Foundry® documentation.

If you are using an SSL certificate signed by an internal root Certificate Authority (CA), you must use a custom Java buildpack whose keystore contains the internal root CA. Follow the below steps to create and add the custom buildpack.

1. Create a fork of the Cloud Foundry Java buildpack.
2. Add your `cacerts` file to `resources/open_jdk_jre/lib/security/cacerts`.
3. Following the instructions in the Adding Buildpacks to Cloud Foundry topic, add your custom buildpack to Pivotal Cloud Foundry®.
4. Using the cf CLI as discussed in the Platform and Product Requirements section above, verify that the custom buildpack is now the default Java buildpack.

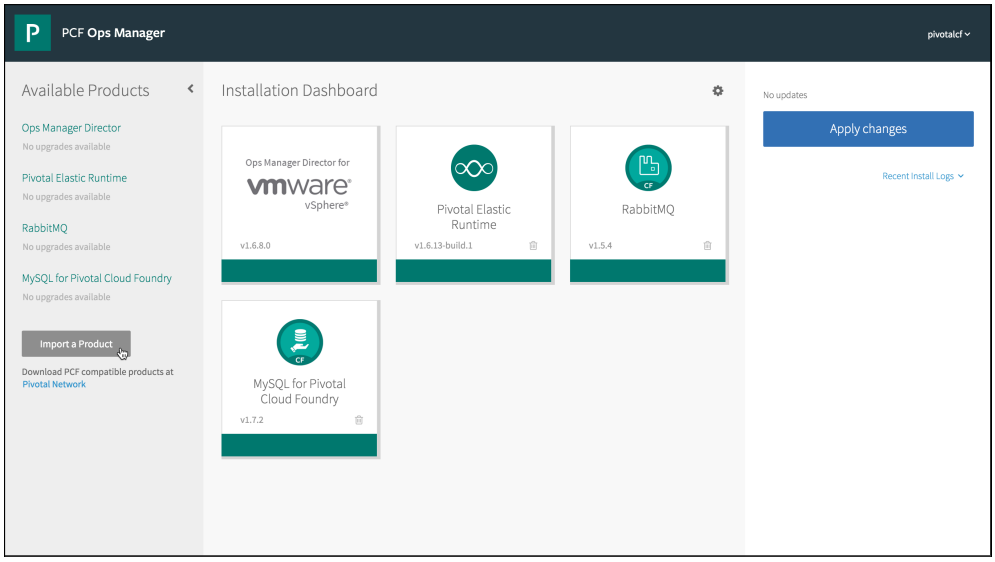
Installing Spring Cloud Services for Pivotal Cloud Foundry®

Page last updated:

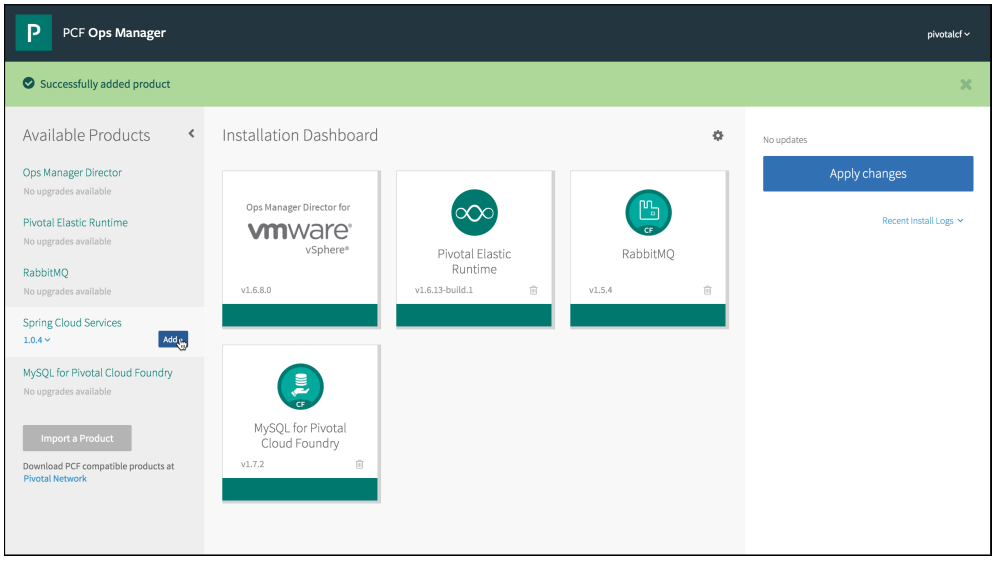
Make sure that you have or have completed all products and requirements listed in the Prerequisites subtopic. Then follow the below steps to install Spring Cloud Services.

Installation Steps

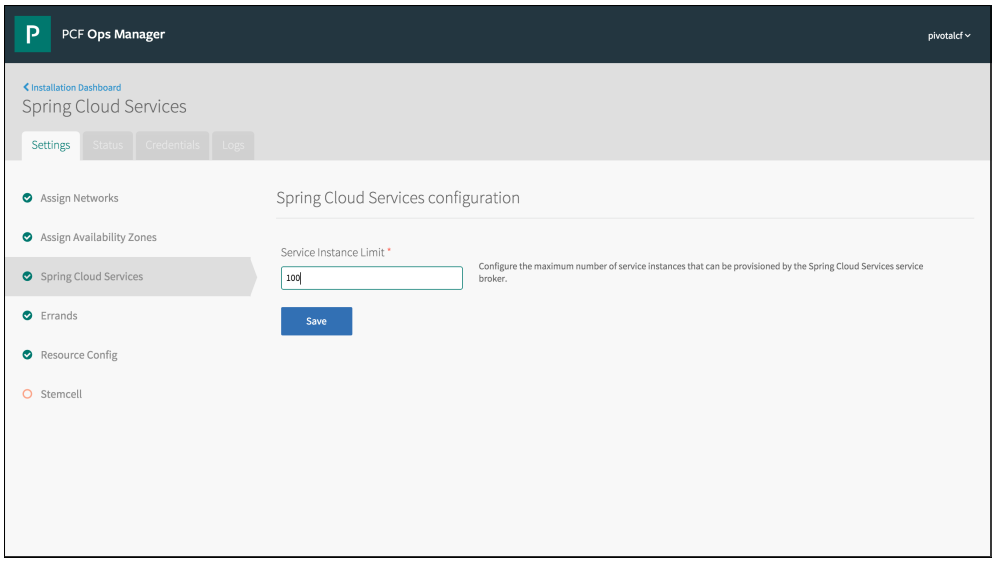
1. Download Spring Cloud Services from Pivotal Network.
2. In Pivotal Cloud Foundry® (PCF) Operations Manager, click **Import a Product** on the left sidebar to upload the `p-spring-cloud-services-<version>.pivotal` file.



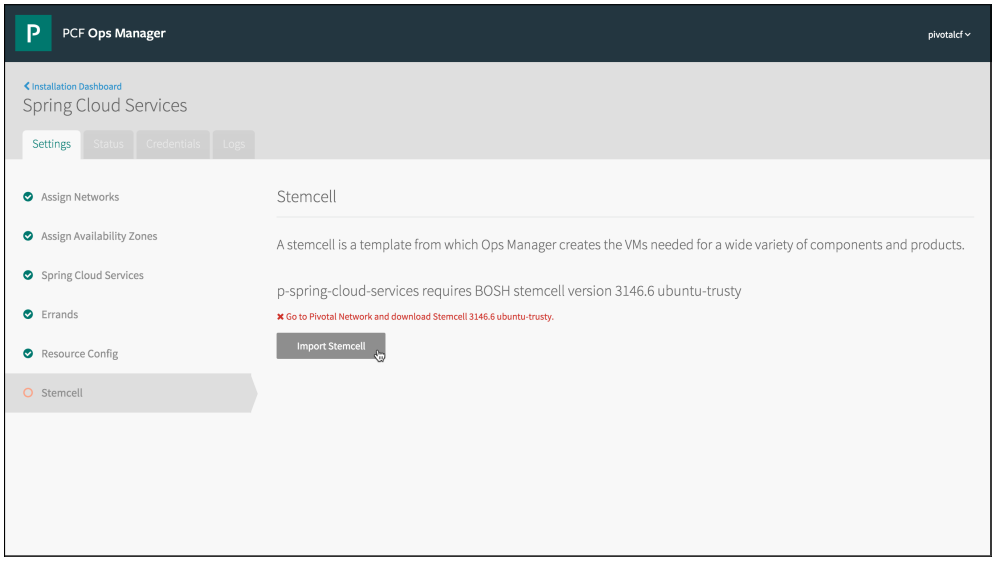
3. Hover over **Spring Cloud Services** in the **Available Products** list and click the **Add** » button.



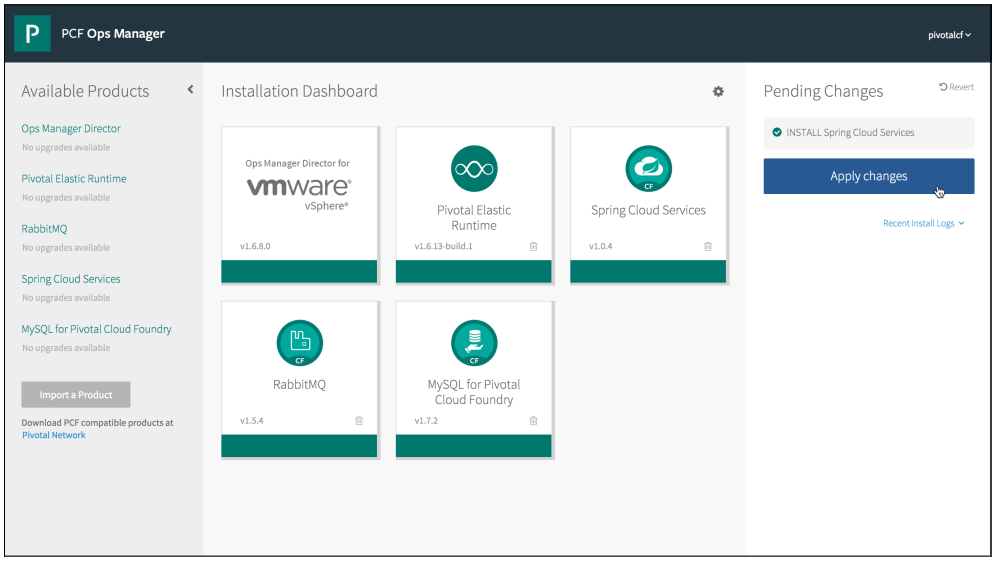
4. When the **Spring Cloud Services** tile appears in the **Installation Dashboard**, click it. In the **Settings** tab, click **Spring Cloud Services** to configure the **Service Instance Limit** setting. The value of this setting is the maximum number of service instances that the Spring Cloud Services 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.



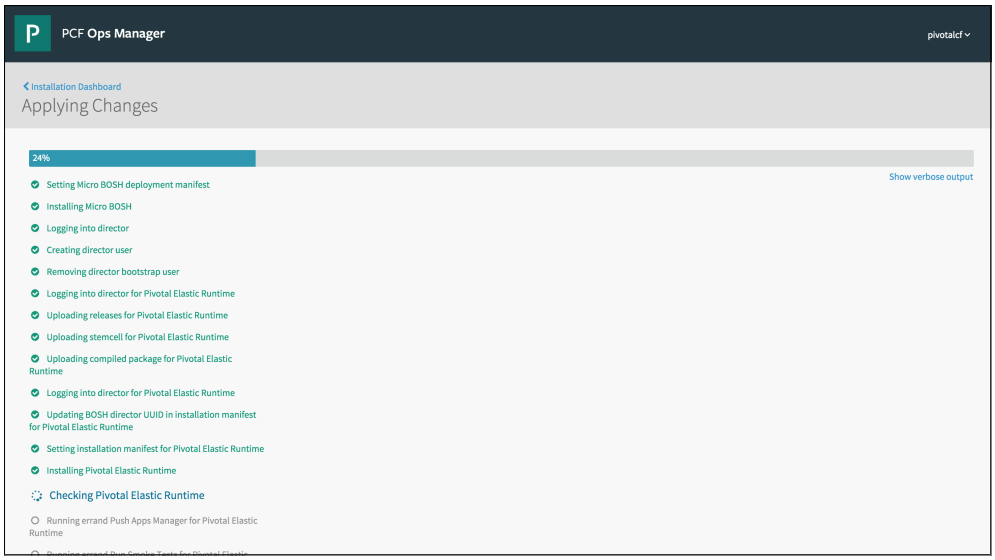
5. If the **Stemcell** tab is highlighted in orange, click it. Download the correct stemcell from Pivotal Network and click **Import Stemcell** to import it.



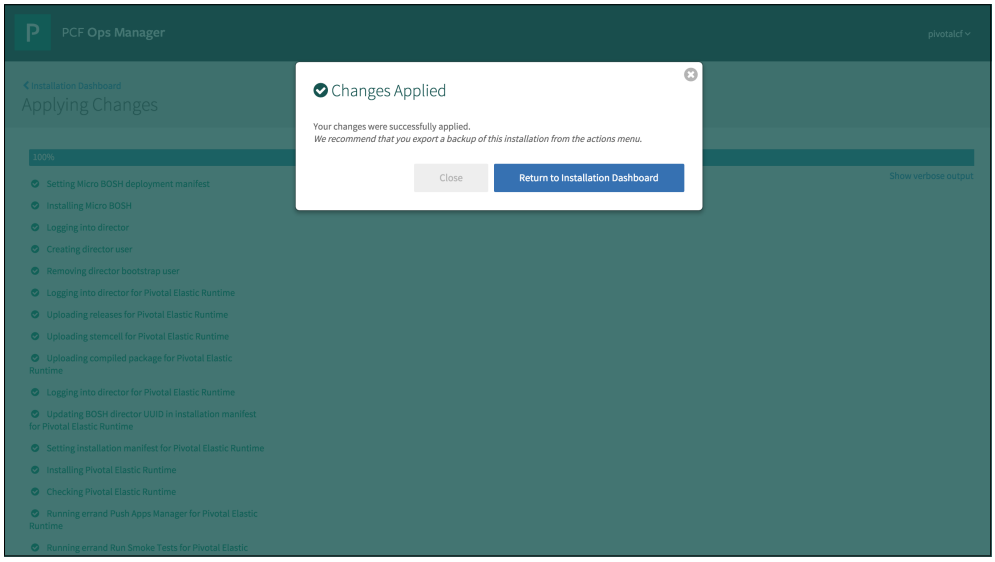
6. Start the installation process by clicking **Apply changes** on the right sidebar.



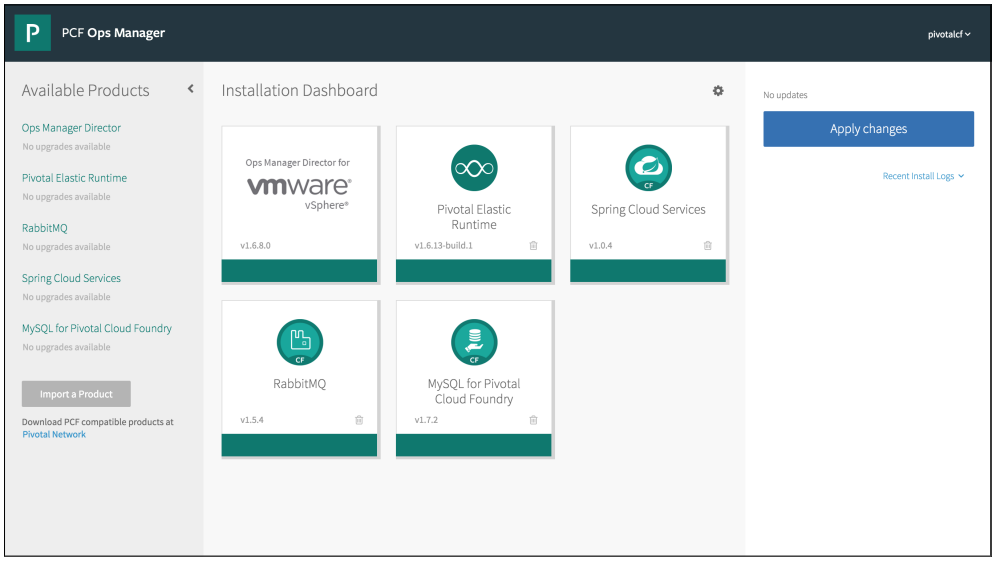
7. The installation process may take 20 to 30 minutes.



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



9. 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.

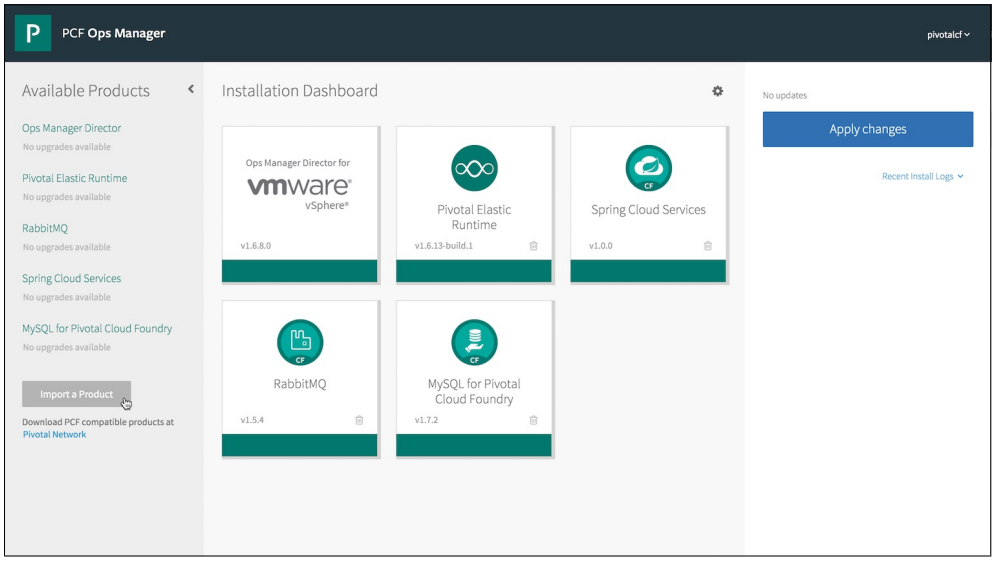
Upgrades on Pivotal Cloud Foundry® 1.7

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.

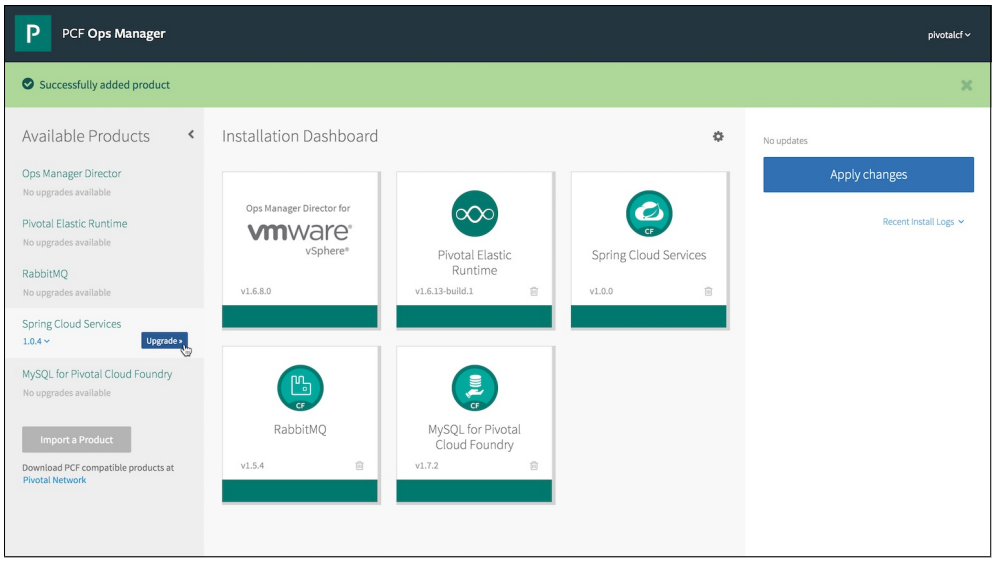
Product Upgrade Steps

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

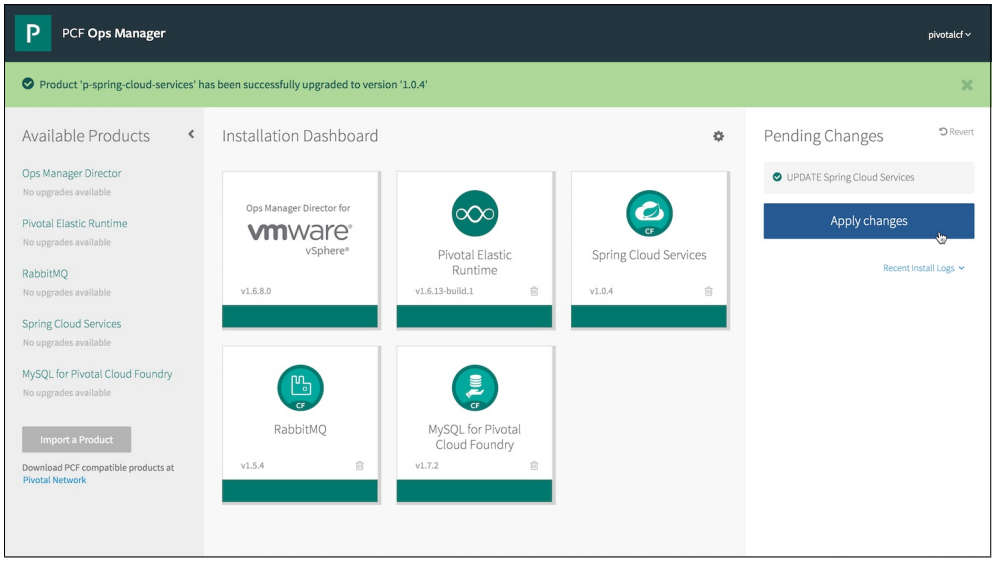
1. Download the latest version of Spring Cloud Services from Pivotal Network.
2. In Pivotal Cloud Foundry® (PCF) Operations Manager, click **Import a Product** on the left sidebar to upload the `p-spring-cloud-services-<version>.pivotal` file.



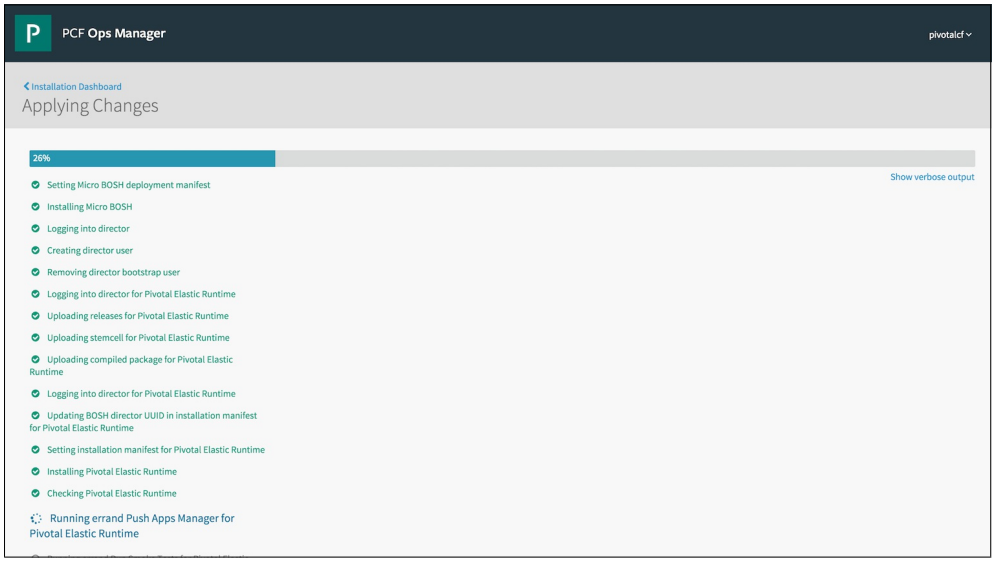
3. After uploading the file, hover over **Spring Cloud Services** in the **Available Products** list and click the **Upgrade »** button.



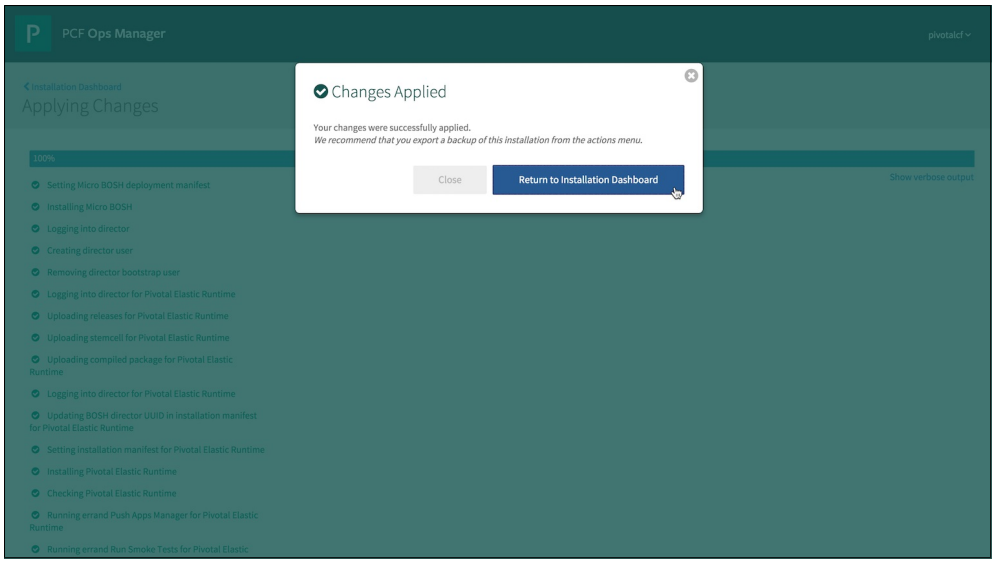
4. Click **Apply changes** on the right sidebar.



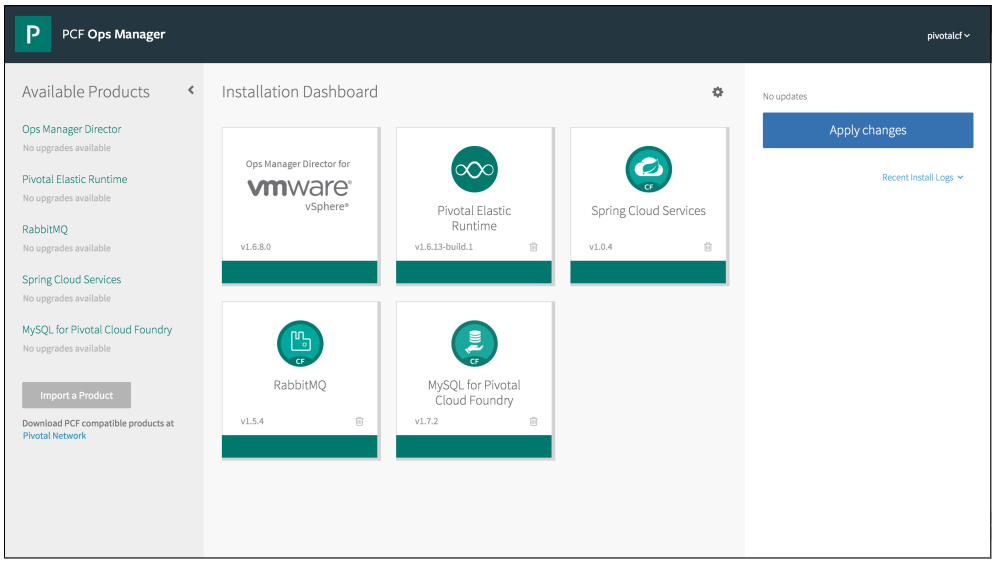
5. The upgrade process may take 20 to 30 minutes.



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



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



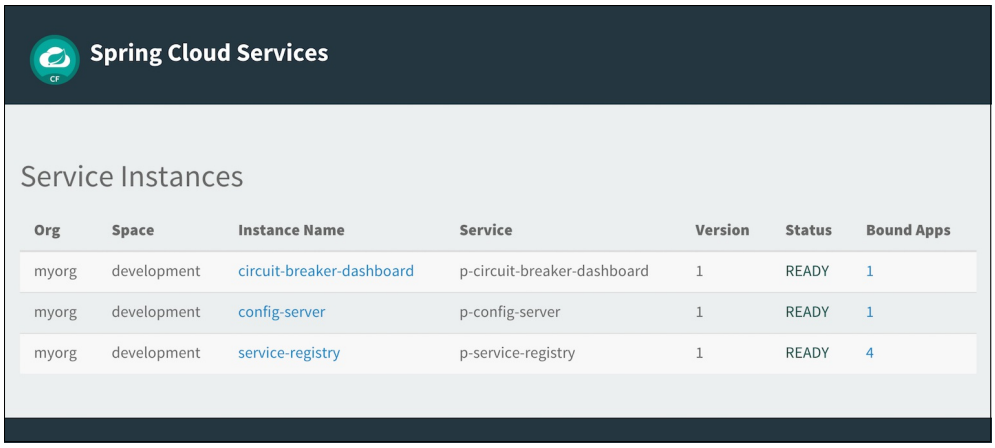
Service Instance Upgrade Steps

Note: Use of `cf update-service` to upgrade service instances is available only in Spring Cloud Services 1.0.4 and later.

Important: In the current version of Spring Cloud Services, upgrading a service instance will result in downtime for the instance while the upgrade is completed.

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

1. To see the current version of a service instance, visit the Service Instances Dashboard. (See the Service Instances Dashboard section of the Operator Information subtopic.)

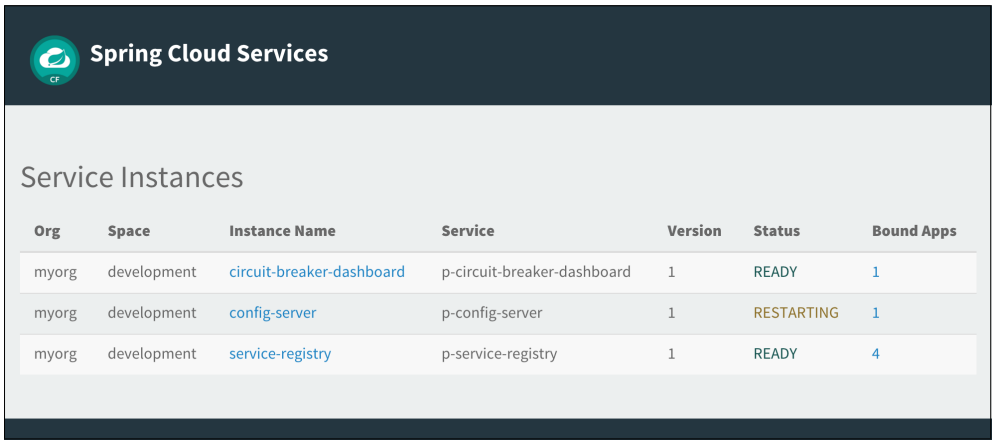


2. Run `cf update-service -c '{"upgrade":true}' SERVICE_NAME`, where `SERVICE_NAME` is the name of a service instance:

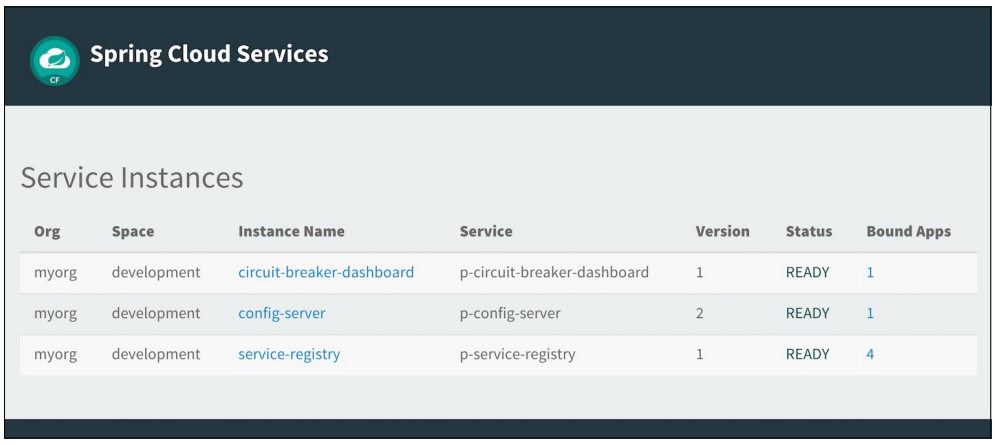
```
$ cf update-service -c '{"upgrade":true}' config-server
Updating service instance config-server as user...
OK
```

Note: The `cf update-service` command begins the upgrade process. The actual upgrade may not have finished when the command exits.

3. Wait while any backing applications belonging to the service instance are upgraded. The Dashboard will show the `RESTARTING` status for the instance.



4. When the service instance has been upgraded, the Dashboard will show the new version.



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 Starters

To work with Spring Cloud Services service instances, your client application must include the `spring-cloud-services-starter-parent` BOM.

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

```
<parent>
 <groupId>io.pivotal.spring.cloud</groupId>
 <artifactId>spring-cloud-services-starter-parent</artifactId>
 <version>1.0.2.RELEASE</version>
 <relativePath/> <!-- lookup parent from repository -->
</parent>
```

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:0.5.6.RELEASE")
  }
}

apply plugin: "io.spring.dependency-management"

dependencyManagement {
  imports {
    mavenBom "io.pivotal.spring.cloud:spring-cloud-services-starter-parent:1.0.2.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")
}
```

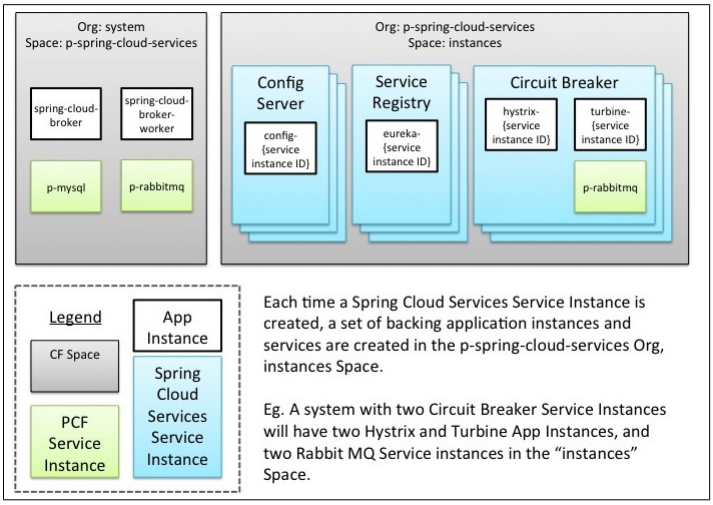
Operator Information

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

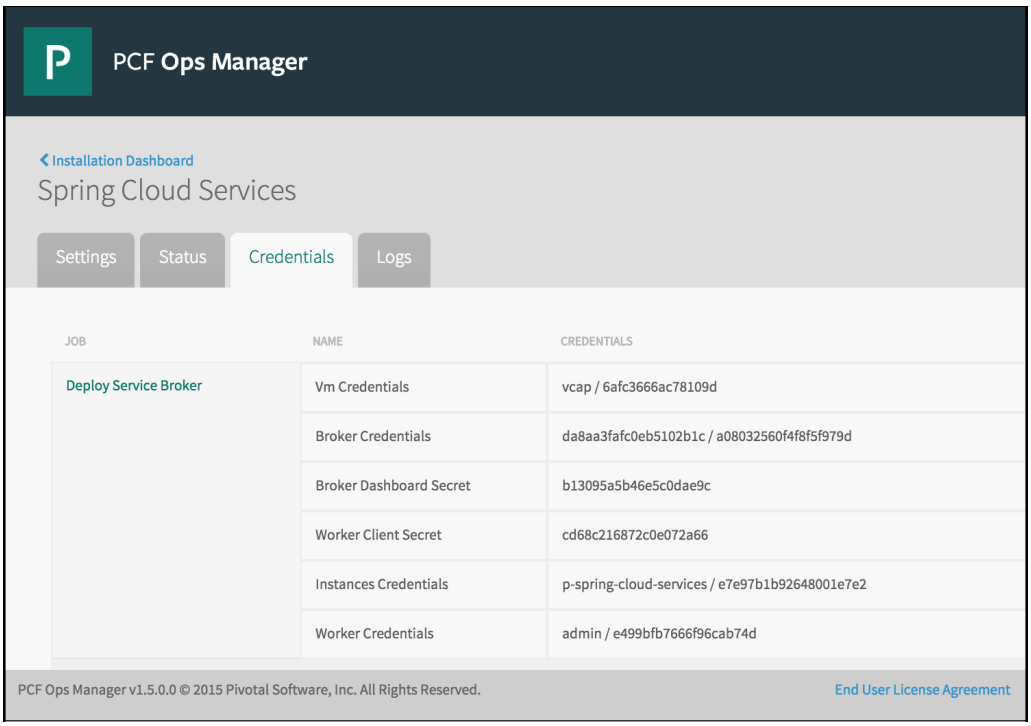
The service broker's functionality is divided between the following two Spring Boot applications, which are deployed in the "system" org to the "p-spring-cloud-services" space.

- 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, copy the Broker Credentials.



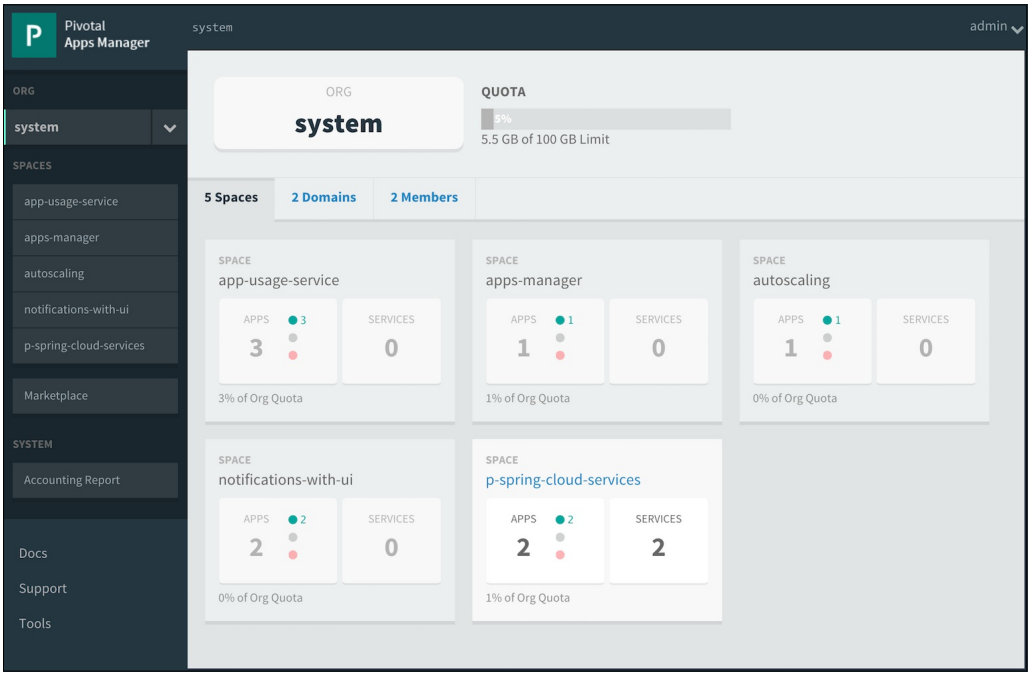
Broker Upgrades

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.

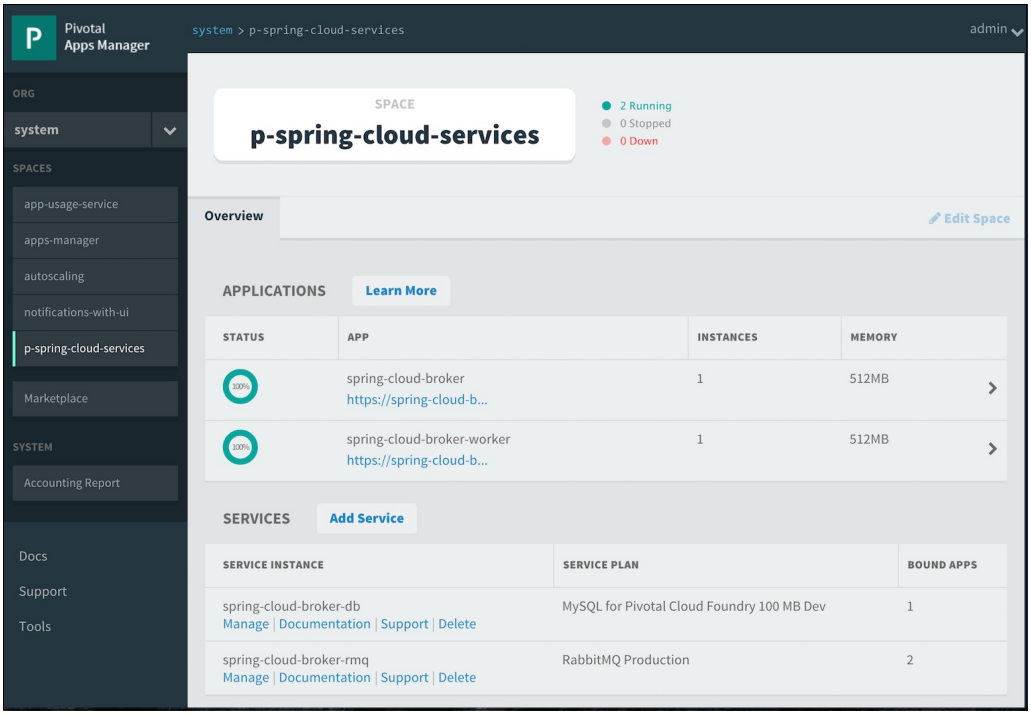
The SB application and the Worker application are deployed using a blue-green deployment strategy. During an upgrade of the service broker, the broker will continue processing requests to provision, deprovision, bind, and unbind service instances, without downtime.

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.

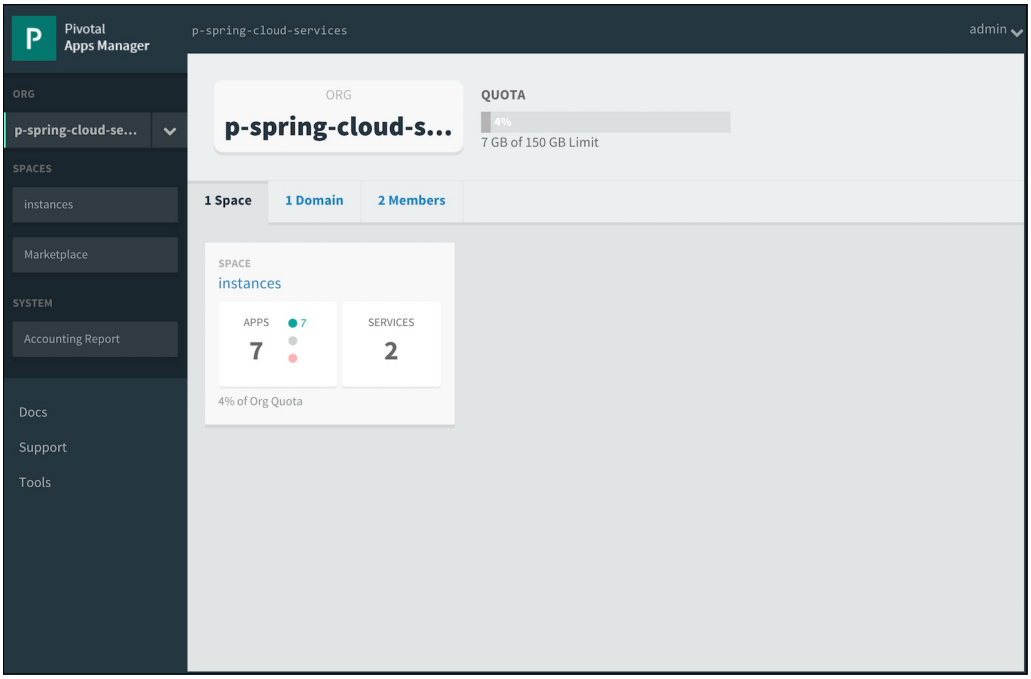
Target the org and space of the service instance.

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

API endpoint:   https://api.wise.com (API version: 2.41.0)
User:           admin
Org:            myorg
Space:          outer

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

name            service           plan       bound apps    last operation
config-server   p-config-server   standard   cook          create succeeded
[...]
```

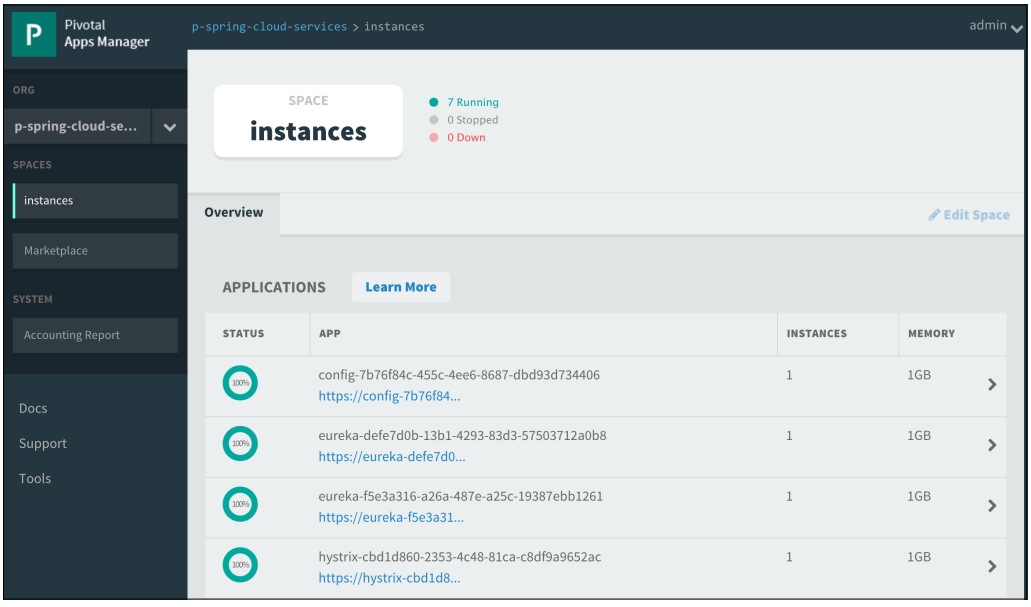
Run `cf service` with the `CF_TRACE` environment variable set to `true`. Copy the value of `resources.metadata.guid`, which is the service instance GUID.

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

VERSION:
6.11.3-cebadc9

REQUEST: [2015-12-09T20:38:44-06:00]
[...]

RESPONSE: [2015-12-09T20:38:44-06:00]
[...]

{
 "total_results": 1,
 "total_pages": 1,
 "prev_url": null,
 "next_url": null,
 "resources": [
   {
     "metadata": {
       "guid": "7b76f84c-455c-4ee6-8687-dbd93d734406",
[...]
```

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.



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 possess bindings to such 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 Eureka backing application, you can get an access token using the binding credentials of an application that is bound to the service instance. To do this, you must have uaac installed.

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

```
$ cf services
Getting services in org myorg / space outer 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 outer 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-7893d5ec-23e9-4fc3-a7ba-484e6c19855b",
     "client_secret": "DWiPkHI83ZVg",
     "uri": "https://config-7b76f84c-455c-4ee6-8687-dbd93d734406.apps.wise.com"
    },
[...]
```

2. From the service instance's entry in the application's `VCAP_SERVICES` environment variable, copy the URL in `credentials.access_token_uri`. Remove the `/oauth/token` path from that URL and target it using `uaac target`:

```
$ uaac target https://p-spring-cloud-services.uaa.cf.wise.com

Target: https://p-spring-cloud-services.uaa.cf.wise.com
```

3. From the service instance's entry in the application's `VCAP_SERVICES` environment variable, copy the values of `credentials.client_id` and `credentials.client_secret`. Run `uaac token client get`, providing those values:

```
$ uaac token client get p-config-server-7893d5ec-23e9-4fc3-a7ba-484e6c19855b -s DWiPkHI83ZVg

Successfully fetched token via client credentials grant.
Target: https://p-spring-cloud-services.uaa.cf.wise.com
Context: p-config-server-7893d5ec-23e9-4fc3-a7ba-484e6c19855b, from client p-config-server-7893d5ec-23e9-4fc3-a7ba-484e6c19855b
```

4. Run `uaac context` to display the current context:

```
$ uaac context

[2]*[https://p-spring-cloud-services.uaa.cf.wise.com]
  skip_ssl_validation: true

  [0]*[p-config-server-7893d5ec-23e9-4fc3-a7ba-484e6c19855b]
     client_id: p-config-server-7893d5ec-23e9-4fc3-a7ba-484e6c19855b
     access_token: eyJhbGciOiJSUzI1NiJ9.eyJqdGkiOiIwNWI1NWM5ZS1iZTdlLTQ5YzMtYTBhYi1lMDI5OWE2MDMyM2UiLCJzdWIiOiJwLWNvbmZpZy1zZX_a-bUNcH1ofOTH
     token_type: bearer
     expires_in: 43199
     scope: p-config-server.7b76f84c-455c-4ee6-8687-dbd93d734406.read
     jti: 05b55c9e-be7e-49c3-a0ab-e0299a60323e
```

   Copy the value of `access_token`.

With the access token, you can now make requests directly against the service instance, e.g. using curl. Use the URL in `credentials.uri` from the service instance's entry in the application's `VCAP_SERVICES` environment variable.

For a Config Server service instance:

```
$ curl -H 'Authorization: bearer eyJhbGciOiJSUzI1NiJ9.eyJqdGkiOiIwNWI1NWM5ZS1iZTdlLTQ5YzMtYTBhYi1lMDI5OWE2MDMyM2UiLCJzdWIiOiJwLWNvbmZpZy1zZX_a-bUNcH1c
{"name":"cook","profiles":["default"],"label":"master","propertySources":[{"name":"https://github.com/spring-cloud-services-samples/cook-config/cook.properties","source":{"cook.special":"Pi
```

For a Service Registry service instance:

```
$ curl -H 'Authorization: bearer eyJhbGciOiJSUzI1NiJ9.eyJqdGkiOiI4ZjlkMTIxYy1mNjUwLTRhODMtODE0Mi02ZWNlYWYxNTNiMzEiLCJzdWIiOiJwLXNlcnZpY2UtcmVnaXN0cnktYjE

 1
 UP_1_

  COMPANY

    company.apps.wise.com
     COMPANY
    10.68.224.46
    UP
    [...]
```

The Service Instances Dashboard

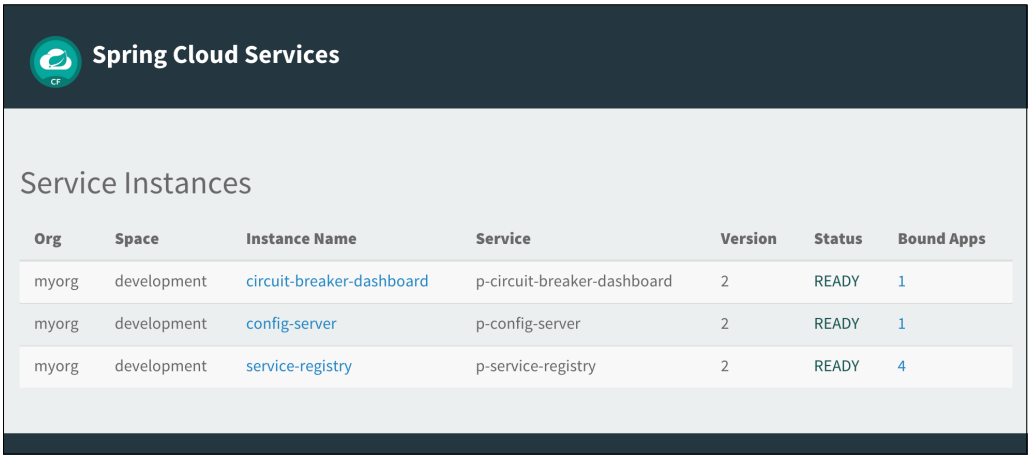
Note: The Service Instances dashboard is available only in Spring Cloud Services 1.0.4 and later.

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.



Click the **Instance Name** of a listed service instance to manage the instance. Hover over the number of an instance's **Bound Apps** to view the names of the applications that are bound to that instance.

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: 2.41.0)
User:           admin
Org:            system
Space:          p-spring-cloud-services

$ 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         512M     1G     spring-cloud-broker.apps.wise.com
spring-cloud-broker-worker   started          1/1         512M     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: 2.41.0)
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-7b76f84c-455c-4ee6-8687-dbd93d734406    started           1/1         1G       1G     config-7b76f84c-455c-4ee6-8687-dbd93d734406.apps.wise.com
```

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

```
$ cf logs config-7b76f84c-455c-4ee6-8687-dbd93d734406
Connected, tailing logs for app config-7b76f84c-455c-4ee6-8687-dbd93d734406 in org p-spring-cloud-services / space instances as admin...
```

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.

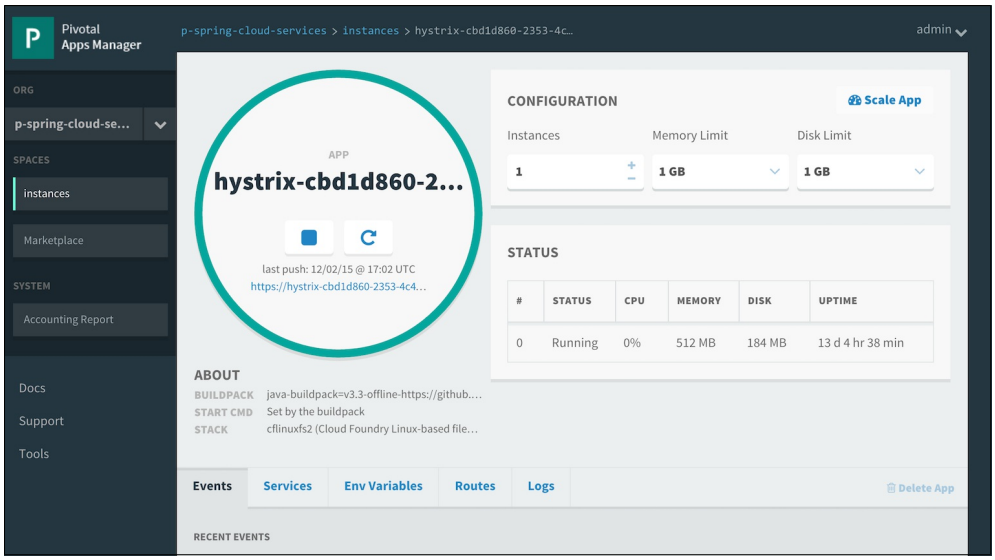
ID	Function
`env`	Displays profiles, properties, and property sources from the application environment
`health`	Displays information on the health and status of the application
`metrics`	Displays metrics information from the application
`mappings`	Displays list of `@RequestMapping` paths in the application
`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.

Endpoints on SB, Worker, Service Registry, or Circuit Breaker Dashboard

The SB application, the Worker application, the Service Registry’s Eureka backing application, and the Circuit Breaker Dashboard’s Hystrix backing application use Dashboard SSO. To access Actuator endpoints on one of these applications, you must be authenticated as a Space Developer in the application’s space and have a current session with the application.

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 or Circuit Breaker Dashboard service instance, access the backing application as described in the Service Instances section.

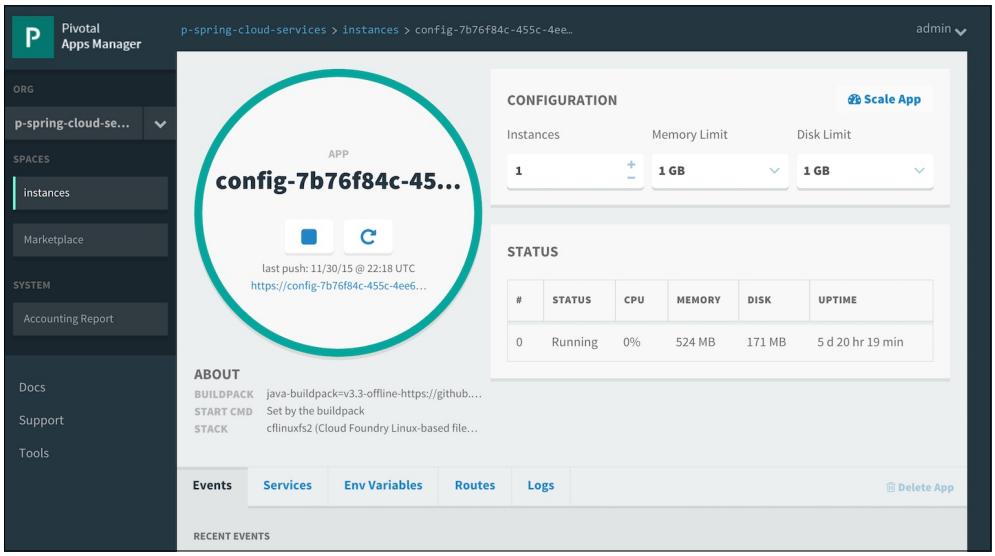


2. Visit the application's URL, appending the endpoint ID to that URL; e.g., for the `health` endpoint on a Circuit Breaker Dashboard service instance's Hystrix backing application, this would be something like `https://hystrix-cbd1d860-2353-4c48-81ca-c8df9a9652ac.apps.wise.com/health`.

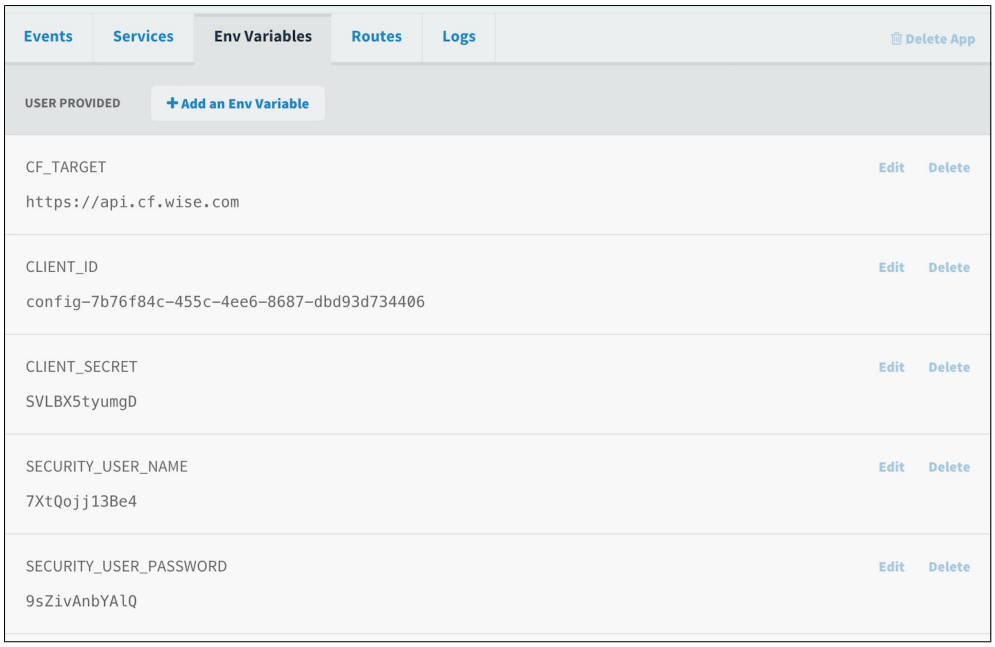
Endpoints on Config Server

The Config Server's Spring Cloud Config Server backing application uses HTTP Basic authentication. To access Actuator endpoints on this application, you must authenticate with the credentials stored in the application's environment variables.

1. Log in to Apps Manager as an admin user and navigate to the relevant application, as described in the Service Instances section.



2. From the **Env Variables** tab, copy the values of the `SECURITY_USER_NAME` and `SECURITY_USER_PASSWORD` environment variables.



3. Visit the application's URL, appending the endpoint ID to that URL; e.g., for the `health` endpoint, this would be something like `config-7b76f84c-455c-4ee6-8687-dbd93d734406.apps.wise.com/health`. When challenged, provide the values of `SECURITY_USER_NAME` and `SECURITY_USER_PASSWORD` from Step 2.

Invoke Config Server 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-7b76f84c-455c-4ee6-8687-dbd93d734406.apps.wise.com/cook/production
   ```

   Or, for the `default` profile:

   ```
   https://config-7b76f84c-455c-4ee6-8687-dbd93d734406.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-7b76f84c-455c-4ee6-8687-dbd93d734406.apps.wise.com/cook/default
   {"name":"cook","profiles":["default"],"label":"master","propertySources":[{"name":"https://github.com/spring-cloud-services-samples/cook-config/cook.properties","source":{"cook.spec
   ```

For a list of path formats that the Config Server accepts, see the Request Paths section of the The Config Server subtopic in the Config Server documentation.

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.

Security Overview for Spring Cloud Services

Glossary

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

- *CC*: The Cloud Foundry Cloud Controller.
- *CC DB*: The Cloud Controller database.
- *Dashboard SSO*: Cloud Foundry's Dashboard Single Sign-On.
- *domain.com*: The value configured by an administrator for the Cloud Foundry system domain.
- *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 HTTPS

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

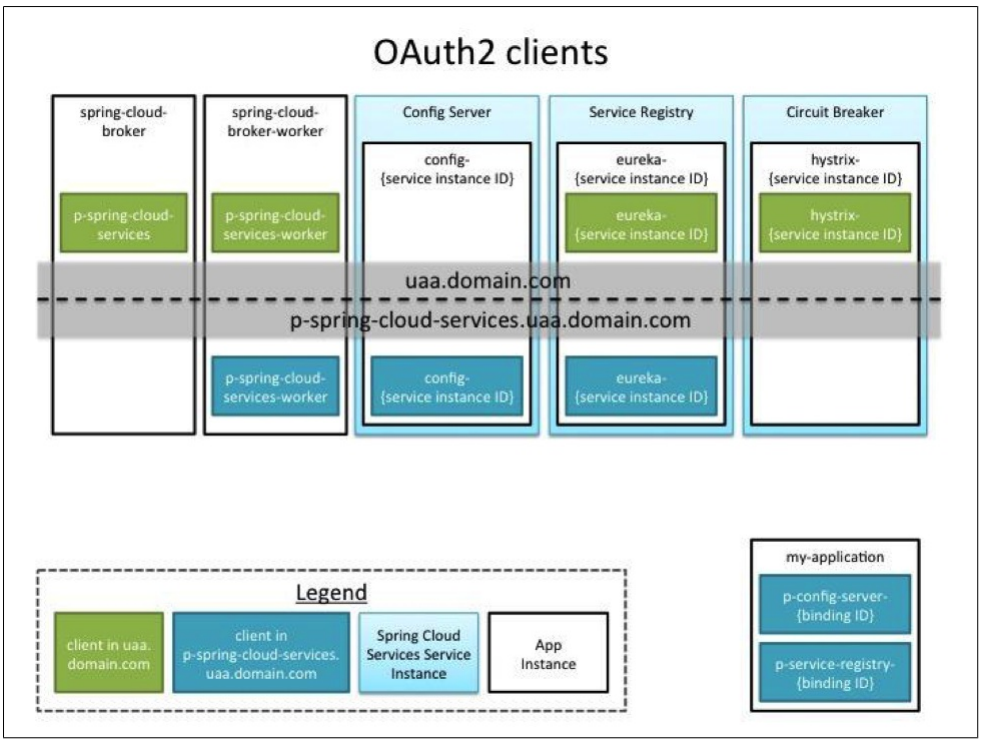
SCS uses the multi-tenancy features of UAA to implement "the principle of least privilege". This is required because SCS creates and deletes OAuth clients when applications bind and unbind to service instances. These clients must also have non-empty authorities. In order to create create clients with 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.

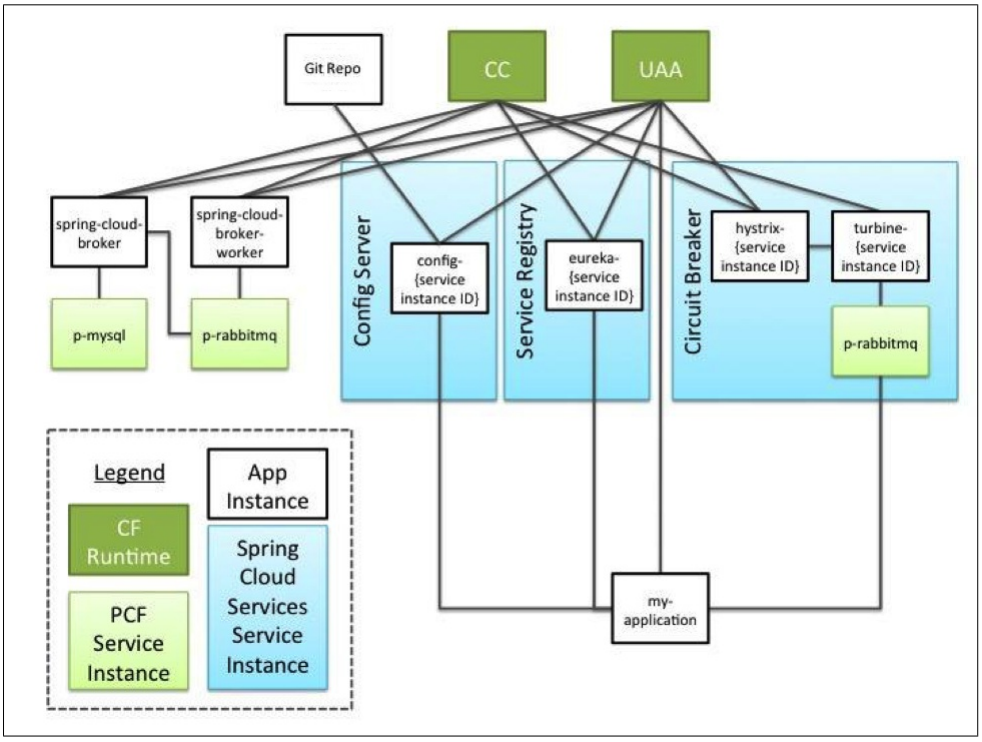
The other Identity Zone is specific to SCS (e.g. `p-spring-cloud-services.uaa.domain.com`). This Identity Zone contains no users, but it contains the clients used by bound applications to access protected resources on SCS service instances. These clients require dynamic authority values, which can only 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 Components

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.

app spring-cloud-broker

The spring-cloud-broker application (the SB) is responsible for receiving Service Broker API requests from the Cloud Controller (CC), and hosts the primary dashboard UI for all service types.

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 is accessed anonymously. 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 spring-cloud-broker-worker (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.

app spring-cloud-broker-worker

The spring-cloud-broker-worker application (the Worker) is responsible for managing service instance backing apps. Some of this work is done asynchronously from the requests that initiate it.

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 app spring-cloud-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 Basic authentication. Credentials that provide this access are set at installation time in the Worker’s environment variables. These environment variables are only accessible to the Operator.

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

app config-server-{guid}

The config-server application is a backing application for Config Server service instances. The GUID in the application name is the service instance ID. This application is responsible for handling requests for configuration values from bound applications.

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. This endpoint can be accessed anonymously, and only returns `{"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: default (12 hours)

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 HTTPS and Basic authentication, with the exception of `/health` as described above. Credentials that provide this access are set at provision time in the config-server application's environment variables. These environment variables are only accessible to the Operator.

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

app eureka-{guid}

The Eureka application is a backing application for Service Registry service instances. The GUID in the application name is the service instance ID. This application hosts a REST API for the Service Registry and a dashboard UI for viewing the status of the Registry.

Entry Points

Eureka REST endpoints

These entry points are defined by Netflix Eureka. Instead of enumerating every endpoint, we will explain the differences in securing the HTTP verbs.

GET

All GET endpoints 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 `p-service-registry-[guid]`
- `iss`: must equal `https://p-spring-cloud-services.uaa.domain.com/oauth/token`
- `scope`: must equal `p-service-registry-[guid].read`

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

POST|PUT|DELETE

All POST, PUT, and DELETE endpoints 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 `p-service-registry-[guid]`
- `iss`: must equal `https://p-spring-cloud-services.uaa.domain.com/oauth/token`
- `scope`: `must equal p-service-registry-[guid].write`

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

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 Application can only be created or modified by a single CF Application. Before continuing, we should define the following Eureka domain objects:

- **Application**: This is identified by appID. CF Applications that register with Eureka will use their `spring.application.name` for this value. When looking up a service, CF Applications will look up Instances via their appID.
- **Instance**: Applications have one or more Instances. CF applications that register with Eureka are actually registering themselves as Instances of an Application. An Application does not exist without Instances; in other words, an Application exists only because an Instance refers to one by appID.

To ensure that a Eureka Application can only be registered by a single CF Application, the Eureka server was modified to record the CF Application's identity, which is conveyed by the `sub` claim of the OAuth 2 Bearer token. This value is saved in the Instance metadata under the key `registrant_principal`. When an Instance is registered (or for any other modification request), if there are other Instances for the same Application, the `registrant_principal` values of the other Instances are checked and must equal the principal of the incoming request; otherwise, the request is rejected.

This scheme could allow multiple CF Applications to register themselves under the same Application ID, if at first there are no other Instances for that Application. In this case, each registration request would be received and handled as if they are the first Instances for that Application. Due to how the `registrant_principal` values are looked up (no locks are placed for performance reasons), this race condition is possible. However, once this state is saved, resulting in multiple `registrant_principal` values for the same Application, any further modification requests made for that application will fail, regardless of who makes the request. The Eureka server will also log this specific error. Because no state modification requests, including heartbeat requests, would be accepted, these Instances would eventually expire.

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**: default (12 hours)

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**: `uaa.resource`
> **Token expiry**: default (12 hours)

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. The credentials are the same as the corresponding client in the UAA Identity Zone.

Circuit Breaker Dashboard Service Instances

app hystrix-{guid}

The Hystrix application is one of the two backing applications for Circuit Breaker Dashboard service instances. The GUID in the application name is the service instance ID. This application is derived from the Netflix Hystrix Dashboard.

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

The Turbine App

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.

app turbine-{guid}

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

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

1.0.45

Release Date: 27th June 2018

Enhancements included in this release:

- The stemcell has been updated to 3468.51. This resolves the following issues:
  - [USN-3658-1]: procps-ng vulnerabilities
  - [USN-3675-1]: GnuPG vulnerabilities
  - [USN-3676-2]: Linux kernel (Xenial HWE) vulnerabilities
  - [USN-3686-1]: file vulnerabilities
  - [USN-3684-1]: Perl vulnerability

1.0.44

Release Date: 4th June 2018

Enhancements included in this release:

- The stemcell has been updated to 3468.46. This resolves the following issues:
  - [USN-3643-1]: Wget vulnerability
  - [USN-3648-1]: curl vulnerabilities
  - [USN-3654-2]: Linux kernel (Xenial HWE) vulnerabilities

1.0.43

Release Date: 10th May 2018

Enhancements included in this release:

- The stemcell has been updated to 3468.42. This resolves the following issues:
  - [USN-3641-1]: Linux kernel vulnerabilities
  - [USN-3631-2]: Linux kernel (Xenial HWE) vulnerabilities
  - [USN-3628-1]: OpenSSL vulnerability
  - [USN-3624-1]: Patch vulnerabilities
  - [USN-3625-1]: Perl vulnerabilities

1.0.42

Release Date: 17th April 2018

Enhancements included in this release:

- The stemcell has been updated to 3468.30. This resolves the following issues:
  - [USN-3619-2]: Linux kernel (Xenial HWE)
  - [USN-3611-1]: OpenSSL
  - [USN-3610-1]: ICU

1.0.41

Release Date: 28th March 2018

Enhancements included in this release:

- The stemcell has been updated to 3468.28. This resolves the following issues:
  - [USN-3586-1]: DHCP
  - [USN-3584-1]: sensible-utils
  - [USN-3598-1]: curl

1.0.40

Release Date: 26th February 2018

Enhancements included in this release:

- The stemcell has been updated to 3468.25. 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

1.0.39

Release Date: 24th January 2018

Enhancements included in this release:

- The stemcell has been updated to 3445.24. This resolves the following issues:
  - [USN-3540-2]: Linux kernel (Xenial HWE) vulnerabilities

1.0.38

Release Date: 19th January 2018

Enhancements included in this release:

- The stemcell has been updated to 3445.23. This resolves the following issues:
  - [USN-3534-1]: GNU C Library vulnerabilities

1.0.37

Release Date: 11th January 2018

Enhancements included in this release:

- The stemcell has been updated to 3445.22. This resolves the following issues:
  - [USN-3522-2]: Linux (Xenial HWE) vulnerability
  - [USN-3522-4]: Linux kernel (Xenial HWE) regression

1.0.36

Release Date: 11th December 2017

Enhancements included in this release:

- The stemcell has been updated to 3445.19. This resolves the following issues:
  - [USN-3505-1]: Linux firmware vulnerabilities

1.0.35

Release Date: 30th November 2017

Enhancements included in this release:

- The previous release used an incorrect version of the stemcell. This release corrects that. The stemcell has been updated to 3445.17.

1.0.34

Release Date: 2nd November 2017

Enhancements included in this release:

- The stemcell has been updated to 3445.16.

1.0.32

Release Date: 18th August 2017

Enhancements included in this release:

- The stemcell has been updated to 3363.31. This resolves the following issues:
  - [USN-3392-2]: Linux kernel (Xenial HWE) regression

1.0.29

Release Date: 15th August 2017

Enhancements included in this release:

- The stemcell has been updated to 3363.30. This resolves the following issues:
  - [USN-3385-2]: Linux kernel (Xenial HWE) vulnerabilities

1.0.28

Release Date: 6th August 2017

Enhancements included in this release:

- The stemcell has been updated to 3363.29. This resolves the following issues:
  - [USN-3378-2]: Linux kernel (Xenial HWE) vulnerabilities

1.0.27

Release Date: 21st June 2017

Enhancements included in this release:

- The stemcell has been updated to 3363.26. This resolves the following issues:
  - [USN-3334-1]: Linux kernel (Xenial HWE) vulnerabilities

1.0.26

Release Date: 5th June 2017

Enhancements included in this release:

- The stemcell has been updated to 3363.25. This resolves the following issues:
  - [USN-3304-1]: Sudo vulnerability

1.0.25

Release Date: 25th May 2017

Enhancements included in this release:

- The stemcell has been updated to 3363.24. This resolves the following issues:
  - [USN-3291-3]: Linux kernel (Xenial HWE) vulnerabilities

1.0.24

Release Date: 27th April 2017

Enhancements included in this release:

- The stemcell has been updated to 3363.20. This resolves the following issues:
  - [USN-3265-2]: Linux kernel (Xenial HWE) vulnerabilities

1.0.23

Release Date: 31st March 2017

Enhancements included in this release:

- The stemcell has been updated to 3363.14. This resolves the following issues:
  - [USN-3249-2]: Linux kernel (Xenial HWE) vulnerability

1.0.22

Release Date: 10th March 2017

Enhancements included in this release:

- The stemcell has been updated to 3263.21. This resolves the following issues:
  - [USN-3220-2]: Linux kernel (Xenial HWE) vulnerability

1.0.21

Release Date: 28th February 2017

Enhancements included in this release:

- The stemcell has been updated to 3263.20. This resolves the following issues:
  - [USN-3208-2]: Linux kernel (Xenial HWE) vulnerabilities

1.0.20

Release Date: 30th January 2017

Enhancements included in this release:

- The stemcell has been updated to 3263.17. This resolves the following issues:
  - [USN-3161-2]: Linux kernel (Xenial HWE) vulnerabilities
  - [USN-3169-2]: Linux kernel (Xenial HWE) vulnerabilities
  - [USN-3172-1]: Bind vulnerabilities

1.0.19

Release Date: 14th December 2016

Enhancements included in this release:

- The stemcell has been updated to 3263.13. This resolves the following issues:
  - [USN-3156-1]: APT vulnerability

1.0.18

Release Date: 7th December 2016

Enhancements included in this release:

- The stemcell has been updated to 3263.12. This resolves the following issues:
  - [USN-3151-2]: Linux kernel (Xenial HWE) vulnerability

1.0.17

Release Date: 21st October 2016

Enhancements included in this release:

- The stemcell has been updated to 3233.3. This resolves the following issues:
  - [USN-3106-2]: Linux kernel (Xenial HWE) vulnerability

1.0.16

Release Date: 12th October 2016

Enhancements included in this release:

- The stemcell has been updated to 3233.2 for stability and bug fixes.

1.0.15

Release Date: 6th October 2016

Enhancements included in this release:

- The stemcell has been updated to 3233.1 as an upgrade to Linux version 4.4 kernel and to enable future upgrades.

1.0.13

Release Date: 28th September 2016

Important: Spring Cloud Services version 1.0.13 was removed from Pivotal Network due to an incorrect stemcell dependency. Please upgrade to version 1.0.14 or later.

Enhancements included in this release:

- The stemcell has been updated to 3232.21. This resolves the following issues:
  - [USN-3087-2]: OpenSSL regression

1.0.12

Release Date: 24th August 2016

This release updates the stemcell.

Features included in this release:

- The stemcell has been updated to 3232.17. This resolves the following issues:
  - [USN-3064-1]: GnuPG vulnerability

1.0.11

Release Date: 1st July 2016

Features included in this release:

- The stemcell has been updated to 3232.12. This resolves the following issues:
  - [USN-3020-1]: Linux kernel (Vivid HWE) vulnerabilities

1.0.10

Release Date: 14th June 2016

Features included in this release:

- The stemcell has been updated to 3232.8. This resolves the following issues:
  - [USN-3001-1]: Linux kernel (Vivid HWE) vulnerabilities

1.0.9

Release Date: 6th May 2016

Features included in this release:

- The stemcell has been updated to 3232.2. This resolves the following issues:
  - [USN-2959-1]: OpenSSL vulnerabilities

1.0.8

Release Date: 18th April 2016

Fixes included in this release:

- The Config Server now applies the values of its “Git Branch” and “Search Paths” configuration form fields when retrieving configuration for an application. These values were not applied from version 1.0.4 of Spring Cloud Services.

1.0.7

Release Date: 16th March 2016

Features included in this release:

- The stemcell has been updated to 3146.10. This resolves the following issues:
  - [USN-2929-1]: Linux kernel vulnerabilities

1.0.6

Release Date: 16th March 2016

Fixes included in this release:

- 1.0.4 was inadvertently released with a version number in its product metadata that prevented upgrades to 1.0.5. This release corrects that.

1.0.5

Release Date: 25th February 2016

Features included in this release:

- The stemcell has been updated to 3146.9. This resolves the following issues:
  - [USN-2910-1]: Linux kernel (Vivid HWE) vulnerabilities
  - [USN-2900-1]: GNU C Library vulnerability
  - [USN-2897-1]: Nettle vulnerabilities
  - [USN-2896-1]: Libgcrypt vulnerability

1.0.4

Release Date: 2nd February 2016

Features included in this release:

- A new Service Instances dashboard is accessible to operators of Spring Cloud Services. The dashboard provides an overview of all service instances and shows their org/space locations, version, status, and bound applications. For more details, see the The Service Instances Dashboard section of the Operator Information.
- Service instances can now be individually upgraded to the latest version using the cf CLI. For more details, see the Service Instance Upgrade Steps section of Upgrades.
- Config Server instance settings (i.e. Git repository configuration) can now be provided using arbitrary parameters via “cf create-service.” They can also be updated via “cf update-service.”
- The stemcell has been updated to 3146.6. This resolves the following issues:
  - [USN-2882-1]: curl vulnerability
  - [USN-2879-1]: rsync vulnerability
  - [USN-2875-1]: libxml2 vulnerabilities
  - [USN-2874-1]: Bind vulnerability

Enhancements included in this release:

- Error pages are now consistent with the Spring Cloud Services UI styles.
- If the version of the Spring Cloud Services service broker has not changed, the Deploy Service Broker errand will not perform a deployment. For more details, see the The Service Broker section of the Operator Information.
- Deployment of the Spring Cloud Services service broker will no longer result in a broker outage. For more details, see theThe Service Broker section of the Operator Information.
- The “cf” and “uaac” commands that are run in BOSH errands are now echoed so that commands can be correlated with their output for troubleshooting purposes.
- Incorrect configuration of generated SSL certificates in Elastic Runtime will now result in an installation failure with a descriptive error message in the install logs. For more details, see the Security Requirements section of Prerequisites.
- Service instance dashboards now include the name of the service instance as breadcrumbs.
- Config Server instances will no longer strip the `{cipher}` prefix from ciphertext contained in the backing Git repository, so that client-side decryption is now possible.
- Pivotal Cloud Foundry (PCF) load balancers with publicly routable IP addresses are now supported.

Fixes included in this release:

- The password field on the Config Server configuration dashboard is now appropriately masked.
- Restart of the backing application for Service Registry service instances will no longer cause a temporary return of empty registry information to clients.
- The Circuit Breaker Dashboard now displays correctly in Safari and Firefox.

1.0.3

Release Date: 7th January 2016

Features included in this release:

- Updated stemcell to 3146.2. This resolves the following issues:
  - [USN-2857-1]: Linux kernel vulnerability
  - [USN-2842-1]: Linux kernel vulnerabilities
  - [USN-2842-2]: Linux kernel (Vivid HWE) vulnerabilities
  - [USN-2836-1]: GRUB vulnerability
  - [USN-2834-1]: libxml2 vulnerabilities

- [USN-2830-1]: OpenSSL vulnerabilities
- [USN-2829-1]: Linux kernel vulnerabilities

1.0.2

Release Date: 1st December 2015

Features included in this release:

- Updated stemcell to 3144. This resolves the following issues:
  - [USN-2815-1]: libpng vulnerabilities
  - [USN-2812-1]: libxml2 vulnerabilities
  - [USN-2810-1]: Kerberos vulnerabilities

1.0.1

Release Date: 12th November 2015

Features included in this release:

- Updated stemcell to 3130. This resolves the following issues:
  - [USN-2806-1]: Linux kernel (Vivid HWE) vulnerability
  - [USN-2798-1]: Linux kernel (Vivid HWE) vulnerabilities
- Removed SVN support from Config Server due to licensing issues

1.0.0

Release Date: 26th October 2015

Features included in this release:

- General Availability release

Known issues:

- Feign clients require Spring Cloud Netflix version 1.0.4. This release of Spring Cloud Services depends on Spring Cloud Netflix version 1.0.3. If you wish to use a Feign client with a Spring Cloud Services service instance, you must manually override the version of Spring Cloud Netflix used by your application.

  Using Gradle:

  ```
  ext["spring-cloud-netflix.version"] = "1.0.4.RELEASE"
  ```

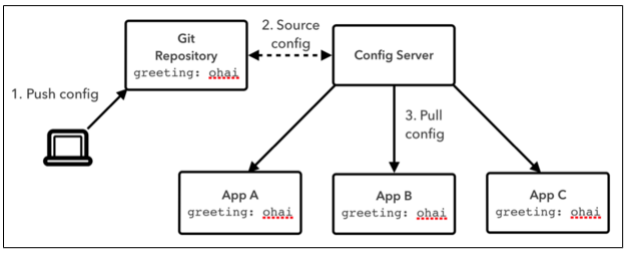
  Using Maven:

  ```
  <properties>
      <!-- override the managed versions of Spring Cloud Netflix dependencies -->
      <spring-cloud-netflix.version>1.0.4.RELEASE</spring-cloud-netflix.version>
  </properties>
  ```

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.

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 topic.

Creating an Instance

Page last updated:

You can create a Config Server instance using either the cf Command Line Interface tool 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.wise.com (API version: 2.43.0)
User:           user
Org:            myorg
Space:          development
```

If desired, 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`. Optionally, you may add the `-c` flag and provide a JSON object that specifies configuration parameters:

- `git.uri`: The URL of the Git repository
- `git.label`: The label
- `git.searchPaths`: The search path or paths
- `git.username`: The username (necessary if the repository is protected by HTTP Basic authentication)
- `git.password`: The password (necessary if the repository is protected by HTTP Basic authentication)

For details on the purpose of each of these fields, see the The Config Server subtopic.

Note: Use of the `-c` flag to specify Config Server settings is available only in Spring Cloud Services 1.0.4 and later.

To create the instance by specifying only the service, plan name, and instance name:

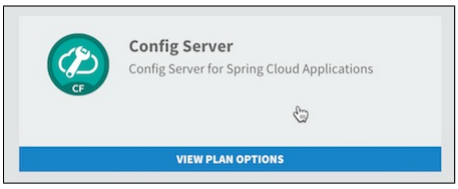
```
$ cf create-service p-config-server standard config-server
Creating service instance config-server in org myorg / space development as user...
OK
```

To create the instance and specify configuration parameters:

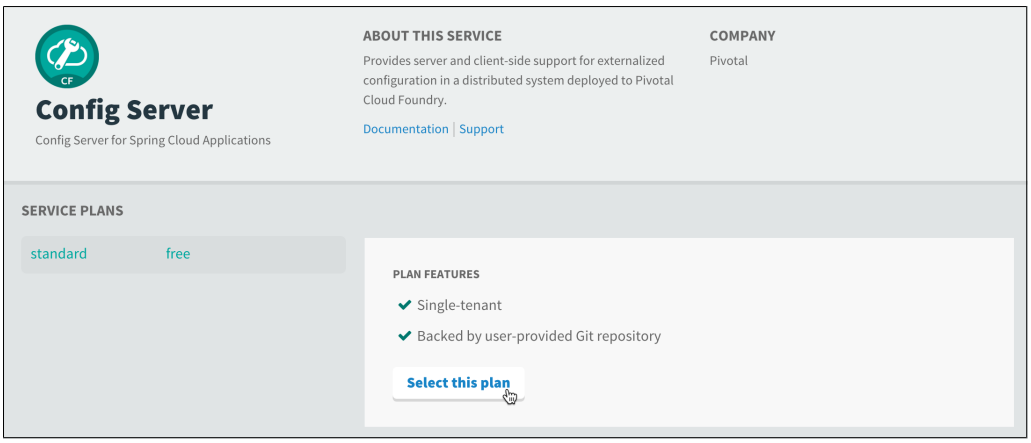
```
$ cf create-service -c '{ "git": { "uri": "https://github.com/spring-cloud-services-samples/cook-config", "label": "master" } }' p-config-server standard config-server
Creating service instance config-server in org myorg / space development as user...
OK
```

Using Apps Manager

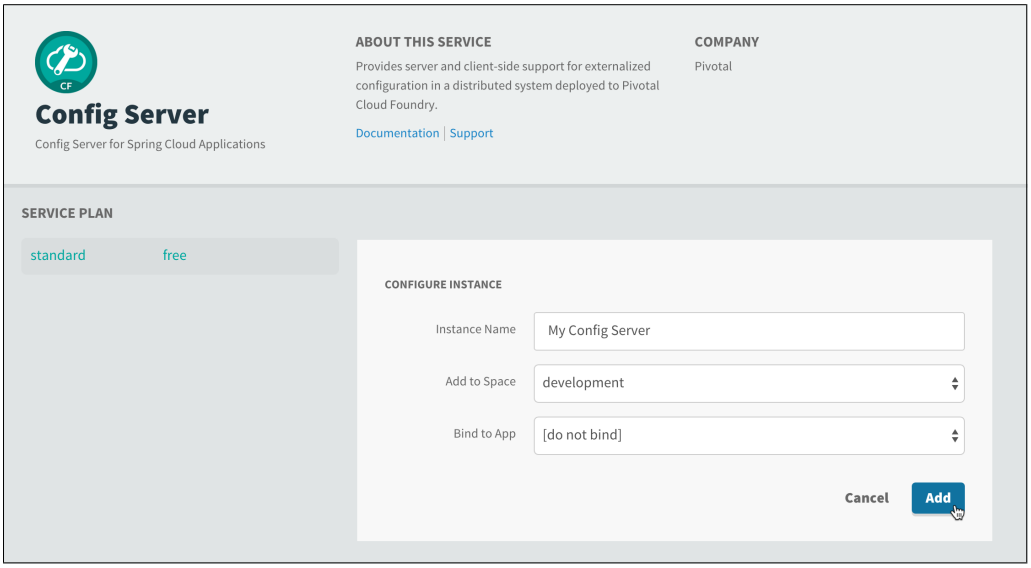
Log into Apps Manager as a Space Developer. In the Marketplace, select **Config Server**.



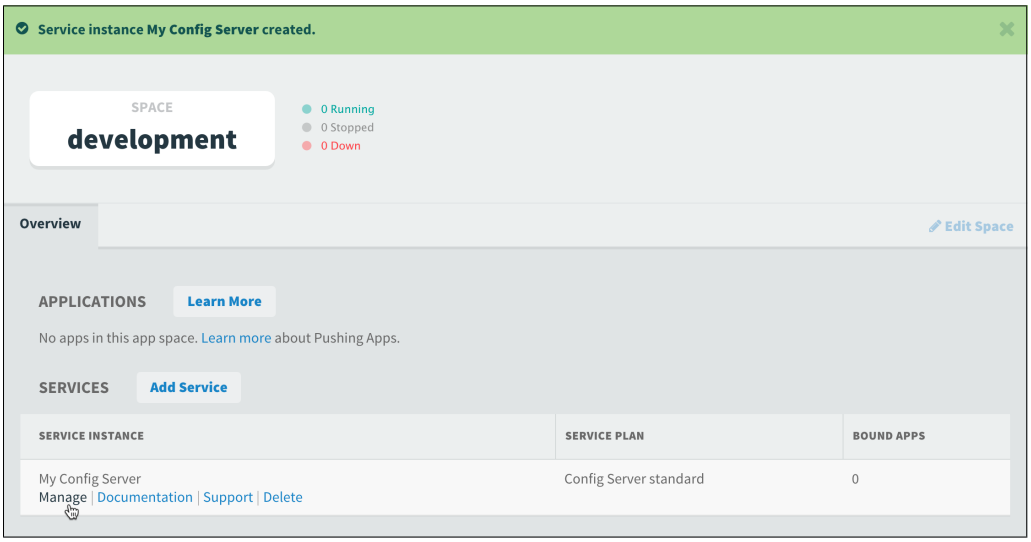
Select the desired plan for the new service.



Provide a name for the service (for example, “My Config Server”). Click the **Add** button.



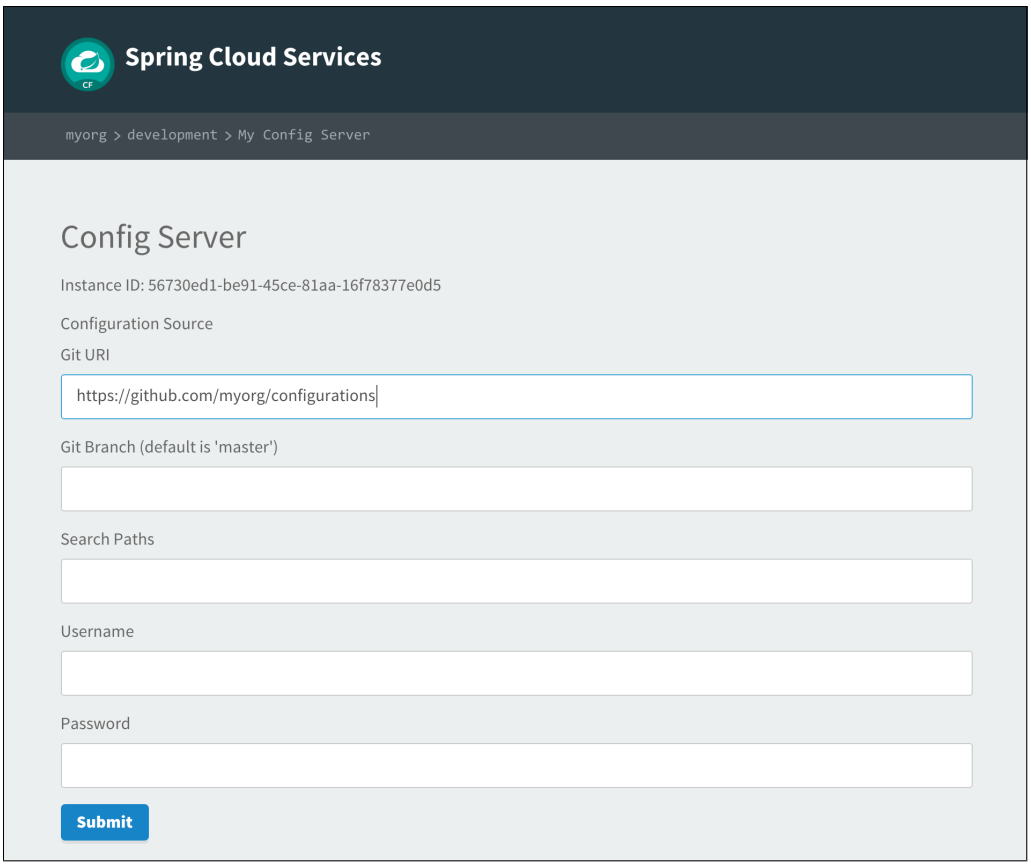
In the **Services** list, click the **Manage** link under the listing for the new service instance.



Enter a repository URI. You can also enter:

- A branch name as the default “label” to be used if the Config Server receives a request without a label.
- A pattern or comma-separated list of patterns which specifies how the Config Server searches for configuration-containing subdirectories in the repository (see the Spring Cloud Config documentation on `searchPaths`).
- A username and password if the repository uses HTTP Basic authentication.

When the appropriate fields are filled out, hit the **Submit** button.



Important: When you submit the form, you will see the message "Config server settings saved", and the Config Server will attempt to initialize with the values that you provided. If the Server cannot initialize, you will get an error: "The Config Server cannot initialize using the configuration that has been provided in this form. Please double-check the configuration, correct any mistakes, and resubmit."

Be sure to double-check the fields that you filled out for accuracy and wait until the Config Server has initialized before proceeding.

The Config Server instance is now ready to be used. For details on how the Config Server stores and retrieves configurations, see the The Config Server subtopic.

Updating Instance Settings

You can update a Config Server instance's settings using either the cf Command Line Interface tool or Pivotal Cloud Foundry® (PCF) Apps Manager.

Using the cf CLI

Note: Use of `cf update-service` to update settings on a Config Server service instance is available only in Spring Cloud Services 1.0.4 and later.

Run `cf update-service` on the service instance and add the `-c` flag with a JSON object that specifies new values for the Config Server's settings:

- `git.uri`: The URL of the Git repository.
- `git.label`: The default "label" to be used if the Config Server receives a request without a label.
- `git.searchPaths`: A pattern or comma-separated list of patterns which specifies how the Config Server searches for configuration-containing subdirectories in the repository (see the Spring Cloud Config documentation on `searchPaths`).
- `git.username`: The username, if the repository is protected by HTTP Basic authentication.
- `git.password`: The password, if the repository is protected by HTTP Basic authentication.

The command should be formatted as in the following example, where `PARAMETERS` is the JSON object and `SERVICE_NAME` is the name of the Config Server service instance:

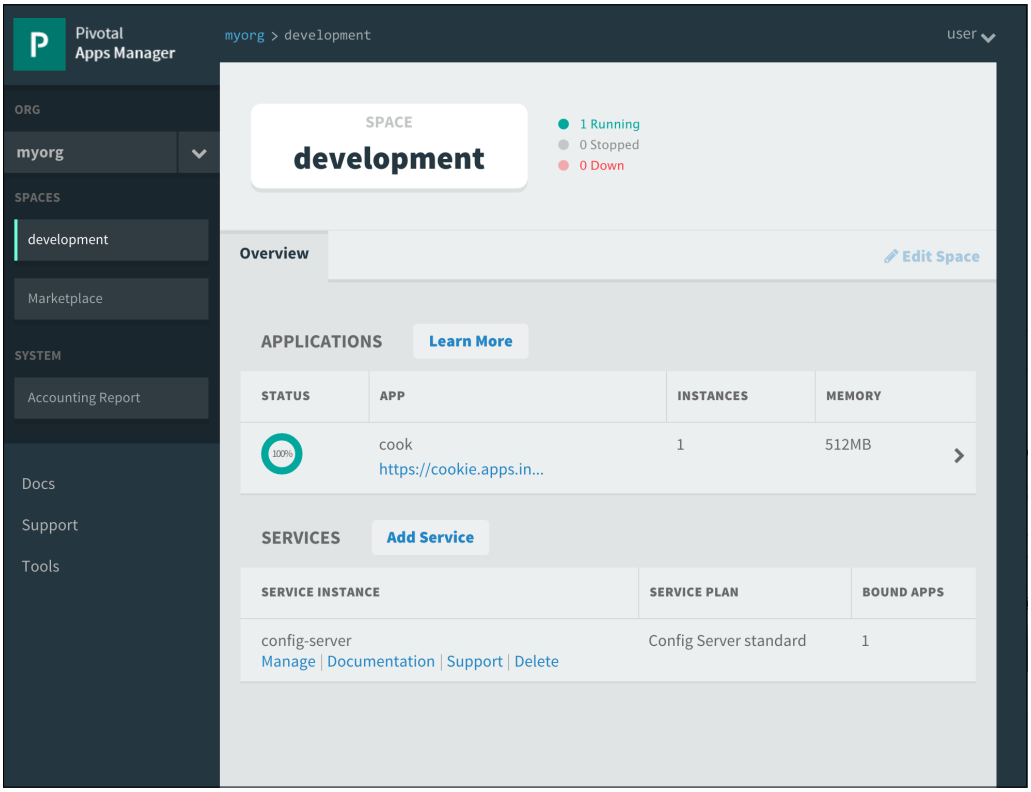
```
cf update-service -c 'PARAMETERS' SERVICE_NAME
```

To update the Git repository URL and label of a Config Server service instance named "config-server", run:

```
$ cf update-service -c '{ "git": { "uri": "https://github.com/spring-cloud-services-samples/cook-config", "label": "master" } }' config-server
Updating service instance config-server as user...
OK
```

Using Apps Manager

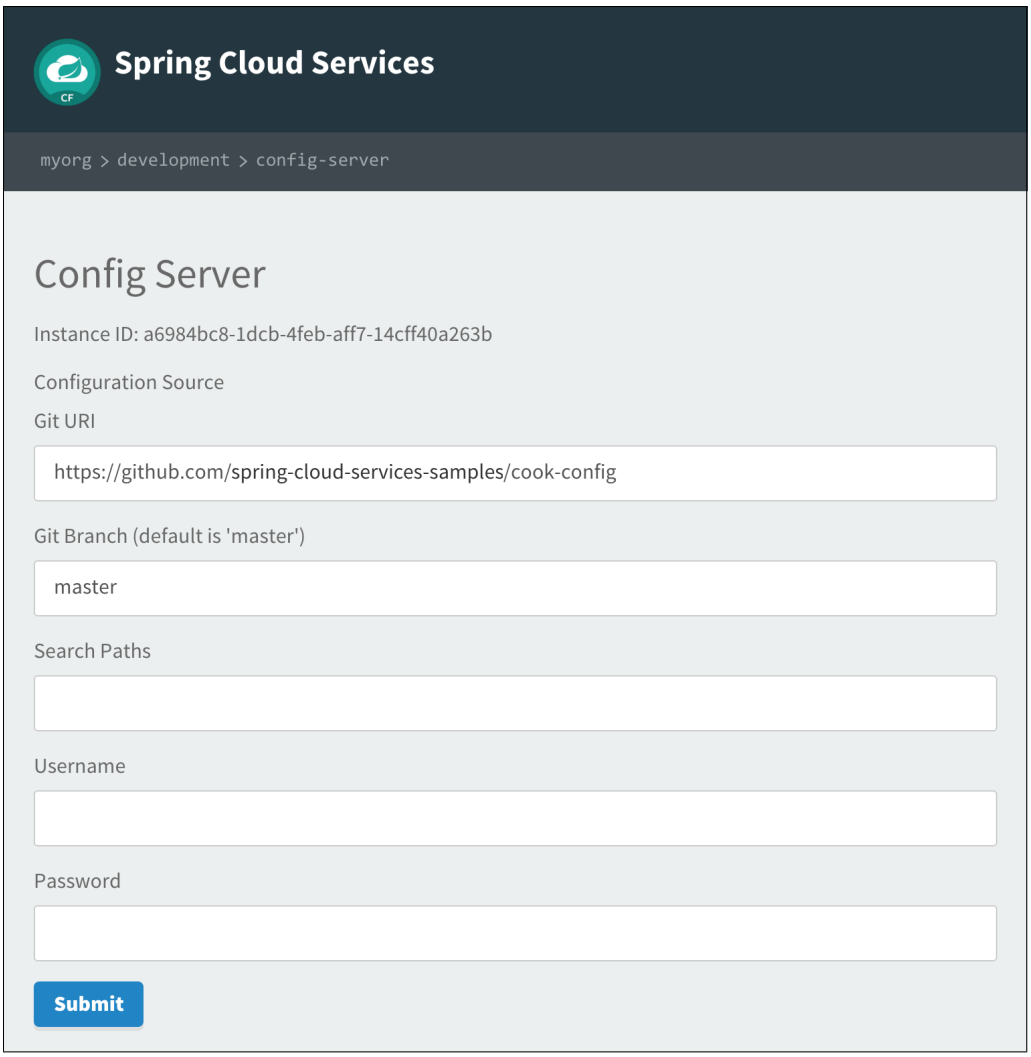
In the **Services** list, click the **Manage** link under the listing for the Config Server service instance.



This will open the Config Server's configuration dashboard, where you can update:

- The URL of the Git repository from which the Config Server will retrieve configurations.
- The default "label" to be used if the Config Server receives a request without a label.
- A pattern or comma-separated list of patterns which specifies how the Config Server searches for configuration-containing subdirectories in the repository (see the Spring Cloud Config documentation on `searchPaths`).
- A username and password if the repository uses HTTP Basic authentication.

When you have finished updating the Config Server's settings, hit the **Submit** button.



Important: When you submit the form, you will see the message "Config server settings saved", and the Config Server will attempt to initialize with the values that you provided. If the Server cannot initialize, you will get an error: "The Config Server cannot initialize using the configuration that has been provided in this form. Please double-check the configuration, correct any mistakes, and resubmit."

Be sure to double-check the fields that you filled out for accuracy and wait until the Config Server has initialized before proceeding.

The Config Server

Page last updated:

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`.

Note: Config Server for Pivotal Cloud Foundry® does not support multiple repositories as configuration sources at this time.

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 Creating an Instance subtopic).

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.

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

Configuration Clients

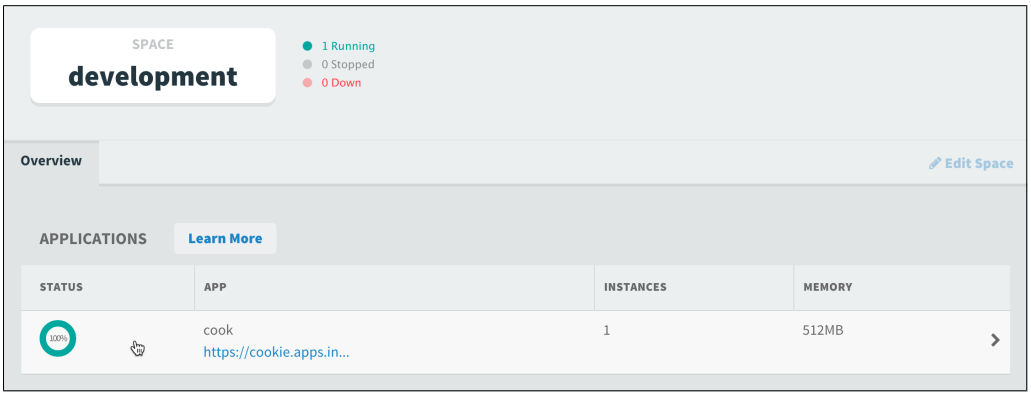
Page last updated:

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.

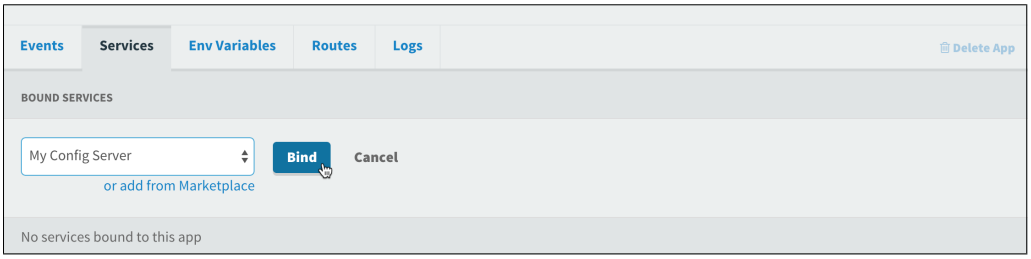
To be given a base URI and client credentials for accessing a Config Server instance, a Cloud Foundry application needs to bind to the instance.

Bind an Application to a Service Instance

Visit the appropriate org and space in Apps Manager, and select an application from the **Applications** list.



In the **Services** tab, click the **Bind a Service** button. Select a service from the dropdown list and click **Bind**.



Alternatively, you can use the cf Command Line Interface tool. Run the command `cf bind-service`, specifying the application name and service name.

```
$ cf bind-service cook My\ Config\ Server
Binding service My Config Server to app cook in org myorg / space development as user...
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 user...
...
```

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 user...
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-153f2832-8f43-48f5-becd-a9cc703dac33",
     "client_secret": "LCfcsfxxC8La",
     "uri": "https://config-56730ed1-be91-45ce-81aa-16f78377e0d5.apps.wise.com"
    },
    "label": "p-config-server",
    "name": "My 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":"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 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 of the The Config Server subtopic). 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 subtopic.

Writing Client Applications

Page last updated:

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

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 `CF_TARGET` environment variable on your client application to the API endpoint of your Elastic Runtime instance:

```
$ cf set-env cook CF_TARGET https://api.cf.wise.com
Setting env variable 'CF_TARGET' to 'https://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
```

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 subtopic). 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 subtopic.)

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 subtopic, 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.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.wise.com",
    "vcap.services.config-server.name":"config-server",
    "vcap.application.uris":"cookie.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.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.wise.com/refresh
["cook.special"]

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

Use Client-Side Decryption

Note: Use of Config Server to serve encrypted configuration values for client-side decryption is available only in Spring Cloud Services 1.0.4 and later.

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 Adding Buildpacks to Cloud Foundry 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 and specify version 1.0.5.RELEASE of Spring Cloud Context in your client application's build file.

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

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

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-context</artifactId>
  <version>1.0.5.RELEASE</version>
</dependency>
```

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

```
compile("org.springframework.security:spring-security-rsa")
compile("org.springframework.cloud:spring-cloud-context:1.0.5.RELEASE")
```

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/HUGhXRdf3KZQ9bqclwmR5vkiLmN9DHlAxS+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.

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.

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.

```
"p-config-server": [
 {
 "credentials": {
  "access_token_uri": "https://p-spring-cloud-services.uaa.cf.wise.com/oauth/token",
  "client_id": "p-config-server-153f2832-8f43-48f5-becd-a9cc703dac33",
  "client_secret": "LCfcsfxxC8La",
  "uri": "https://config-56730ed1-be91-45ce-81aa-16f78377e0d5.apps.wise.com"
 },
 "label": "p-config-server",
 "name": "My 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.

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

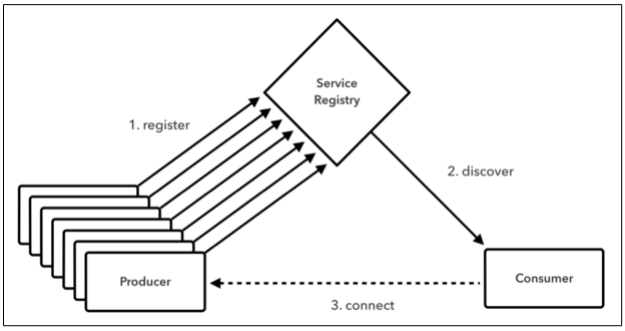
Page last updated:

- 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 topic.

Creating an Instance

Page last updated:

You can create a Service Registry instance using either the cf Command Line Interface tool 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 Config Server product using `cf marketplace -s`.

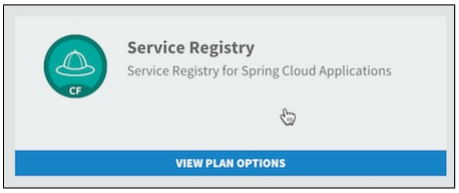
```
$ cf marketplace
Getting services from marketplace in org myorg / space development as user...
OK

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-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.

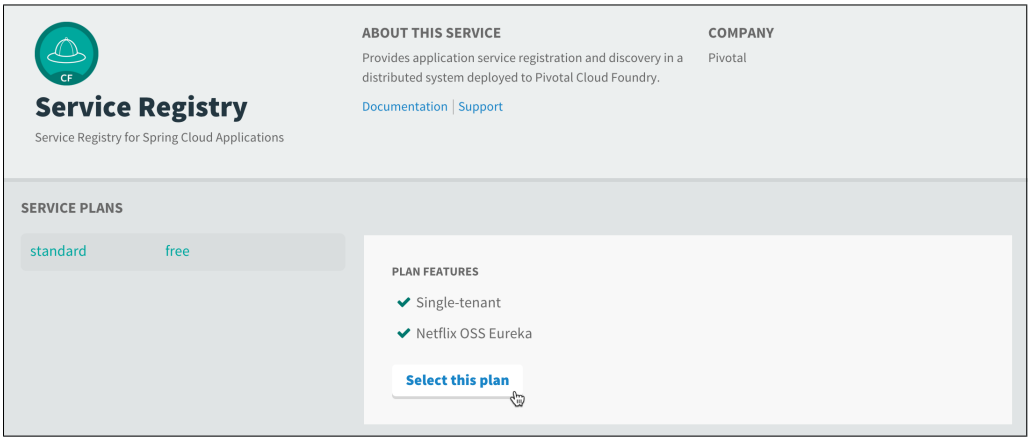
```
$ cf create-service p-service-registry standard My\ Service\ Registry
Creating service instance My Service Registry in org myorg / space development as user...
OK
```

Using Apps Manager

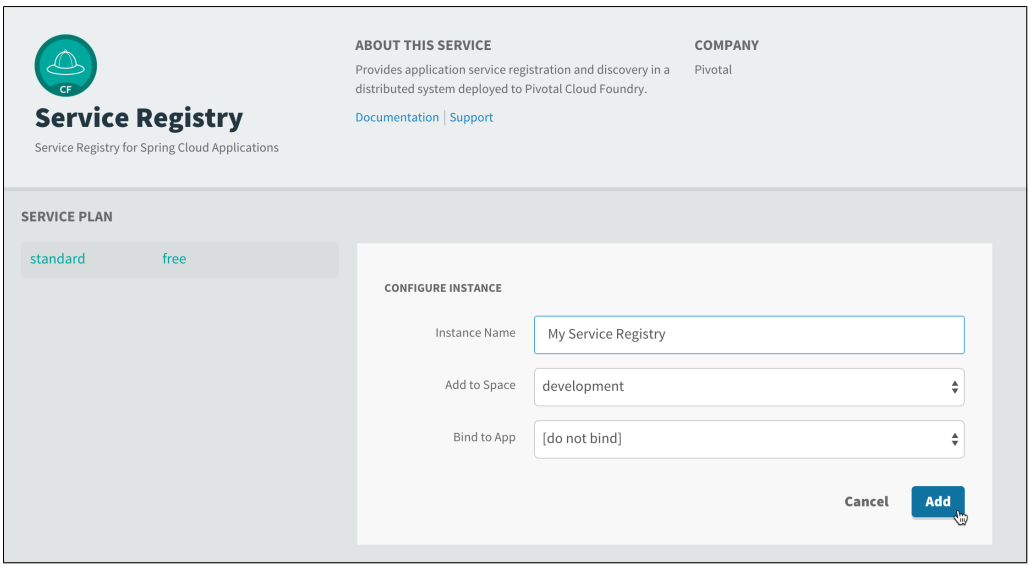
Log into Apps Manager as a Space Developer. In the Marketplace, select **Service Registry**.



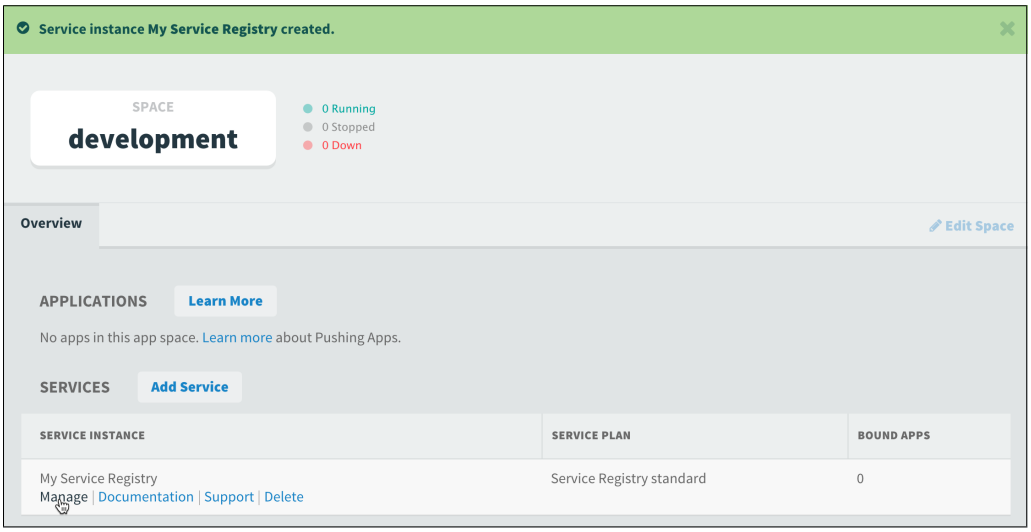
Select the desired plan for the new service.



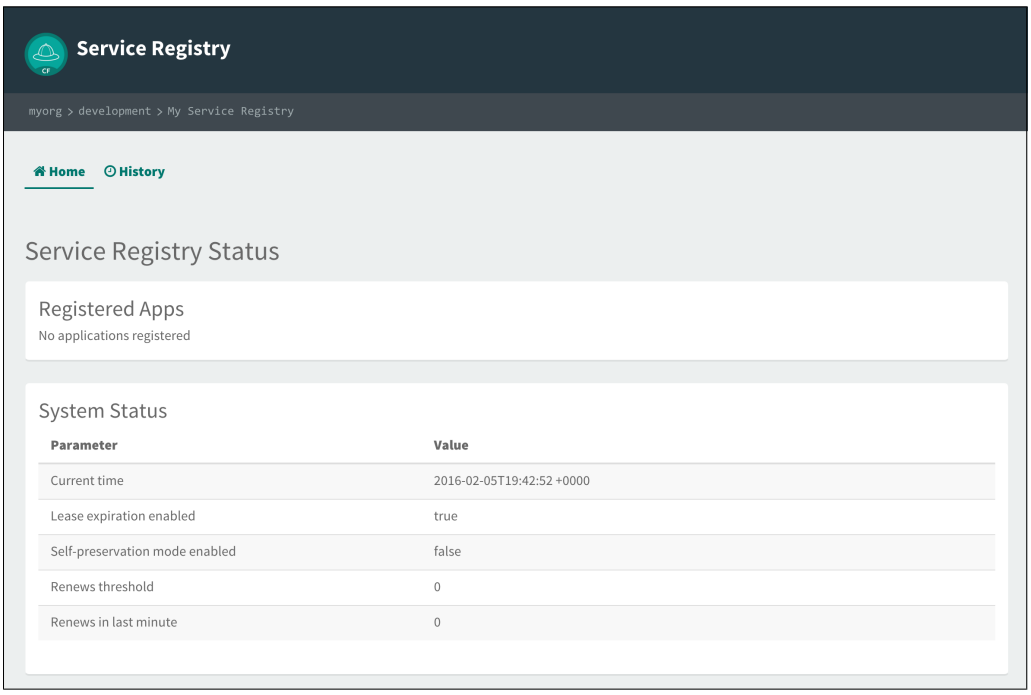
Provide a name for the service (for example, “My Service Registry”). Click the **Add** button.



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; 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 Service Registry instance is now ready to be used. For information about registering a service application with the instance, see the Writing Client Applications subtopic.

Writing Client Applications

Page last updated:

Refer to the sample applications in the "greeting" repository to follow along with the code in this subtopic.

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 dependencies listed in the Client Dependencies topic to your application's build file. Be sure to include the dependencies for Service Registry as well.

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 `CF_TARGET` environment variable on your application to the API endpoint of your Elastic Runtime instance:

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

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

Register a Service

Follow the below instructions to register an application with a Service Registry service instance.

Identify the Application

Your service application must include the `@EnableDiscoveryClient` annotation on a configuration class.

```
@SpringBootApplication
@EnableDiscoveryClient
@RestController
public class MessageGenerationApplication {
  //...
```

The `MessageGenerationApplication` class also has a `/greeting` endpoint which gives a JSON `Greeting` object.

```
  @RequestMapping("/greeting")
  public Greeting greeting(@RequestParam(value="salutation",
                    defaultValue="Hello") String salutation,
                   @RequestParam(value="name",
                    defaultValue="Bob") String name) {
    //...
    return new Greeting(salutation, name);
  }
```

Set the `spring.application.name` property to the value from the application deployment. In `application.yml`:

```
spring:
  application:
    name: ${vcap.application.name}
```

Register the Application

Optionally, you may also specify the registration method that you wish to use for the application, using the `spring.cloud.services.registrationMethod` property. It can take either of two values:

- `route`: The application will be registered using its Cloud Foundry route.
- `direct`: The application will be registered using its host IP and port.

The default value for the `registrationMethod` property is `route`. You may also provide configuration using properties under `eureka.instance`, such as `eureka.instance.hostname` and `eureka.instance.nonSecurePort` (see the Spring Cloud Netflix documentation); if you do, those properties will override the value of `registrationMethod`.

See below for information about the `route` and `direct` registration methods.

Route Registration

An application deployed to Pivotal Cloud Foundry® can have one or more URLs, or "routes," bound to it. If you specify the `route` registration method, the application will be registered with the Service Registry instance using the first of these routes from the `application_uris` list in the application's `VCAP_APPLICATION` environment variable.

```
{
"VCAP_APPLICATION": {
 "application_name": "message-generation",
 "application_uris": [
  "message-generation.wise.com"
 ],
```

Requests from client applications to this registered application go through the Pivotal Cloud Foundry® Router, and the Router provides load balancing between the clients and registered applications.

This registration method is compatible with all deployments of Pivotal Cloud Foundry®.

Direct Registration

An application running on Pivotal Cloud Foundry® is provided with an externally-accessible IP address and port, which are made available in two environment variables: `CF_INSTANCE_IP` and `CF_INSTANCE_PORT`. If you specify the `direct` registration method, the application will be registered with the Service Registry instance using this IP address and port.

Requests from client applications to this registered application will not go through the Pivotal Cloud Foundry® Router. You can provide client-side load balancing using Spring Cloud and Netflix Ribbon.

This registration method is only compatible with Pivotal Cloud Foundry® version 1.5 or higher. In Pivotal Cloud Foundry® Operations Manager, in the **Pivotal Elastic Runtime** tile's **Application Containers**, the "Microservices: Enable cross-container traffic" option must be enabled.

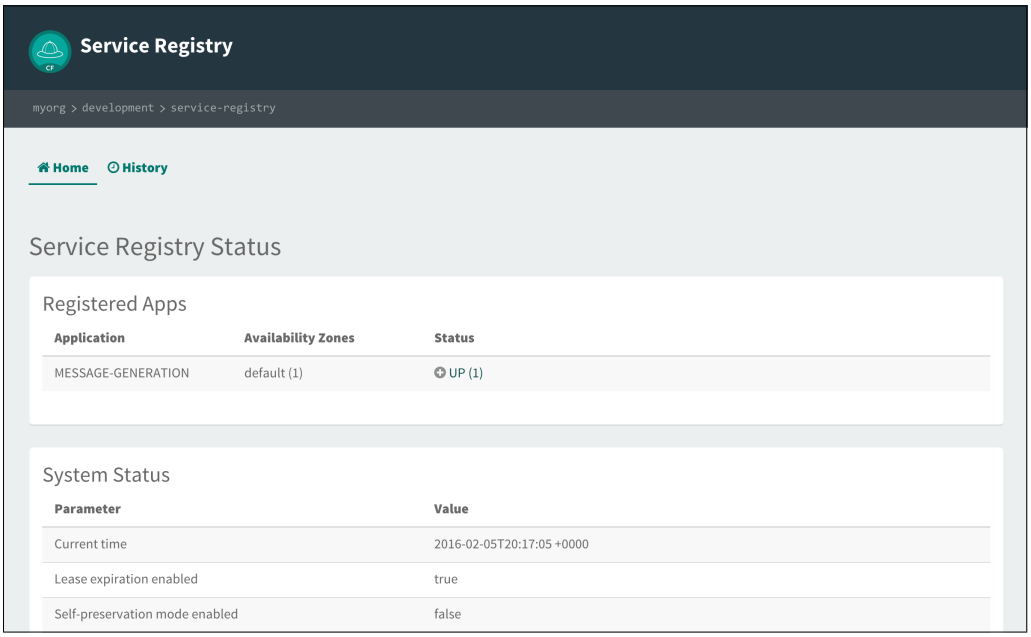
Specify a Registration Method

If you do wish to specify a registration method, set the `spring.cloud.services.registrationMethod` property. In `application.yml`:

```
spring:
  application:
    name: ${vcap.application.name}
  cloud:
    services:
      registrationMethod: route
```

As mentioned above, the `route` method is the default; it will be used if you do not specify `registrationMethod`.

With the `spring.application.name` property set, you can now bind the application to a Service Registry instance. Once it has been bound and restaged and has successfully registered with the Registry, you will see it listed in the Service Registry dashboard (see the Using the Dashboard subtopic).



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 {

  @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.

Important: If your Pivotal Cloud Foundry® installation is configured to only allow HTTPS traffic, you must specify the `https://` scheme in the base URI used by your client application. For the below example, this means that you must change the base URI from `http://message-generation/greeting` to `https://message-generation/greeting`.

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 the Application section above), you have two options:

1. Use the `https://` URI scheme to access the service.
2. In your application configuration, set the `ribbon.IsSecure` property to `false`, and use the `http://` URI scheme to access the service.

If you wish to use HTTP rather than HTTPS, set `ribbon.IsSecure` first. In `application.yml`:

```
ribbon:
  IsSecure: false
```

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 `http://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("http://message-generation/greeting")
      .queryParam("salutation", salutation)
      .queryParam("name", name)
      .build()
      .toUri();

    Greeting greeting = rest.getForObject(uri, Greeting.class);
    return greeting.getMessage();
  }
```

Greeter's `Greeting` class uses Jackson's `@JsonCreator` and `@JsonProperty` to read in the JSON response from Message Generation.

```
  private static class Greeting {

    private String message;

    @JsonCreator
    public Greeting(@JsonProperty("message") String message) {
      this.message = message;
    }

    public String getMessage() {
      return this.message;
    }

  }
```

The Greeter application now responds with a customizable `Greeting` when you access its `/hello` endpoint.

```
$ curl http://greeter.wise.com/hello
Hello, Bob!

$ curl http://greeter.wise.com/hello?name=John
Hello, John!
```

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, you must declare `spring-cloud-starter-feign` as a dependency. Spring Cloud Netflix's Feign support had a bug which was fixed in its version 1.0.4; however, Spring Cloud Services currently depends on Spring Cloud Netflix version 1.0.3. You will therefore also have to manually override the managed versions of the Spring Cloud Netflix dependencies.

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

```
<properties>
  <!-- override the managed versions of Spring Cloud Netflix dependencies -->
  <spring-cloud-netflix.version>1.0.4.RELEASE</spring-cloud-netflix.version>
</properties>

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

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

```
ext["spring-cloud-netflix.version"] = "1.0.4.RELEASE"

dependencies {
    compile("org.springframework.cloud:spring-cloud-starter-feign:1.0.3.RELEASE")
}
```

Your consuming application must include the `@EnableDiscoveryClient` annotation on a configuration class. To have Feign client interfaces automatically configured, it must also use the `@EnableFeignClients` annotation. In the Greeter application, the `GreeterApplication` class contains an `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.

Important: If your Pivotal Cloud Foundry installation is configured to only allow HTTPS traffic, you must specify the `https://` scheme in the base URI used by your client application. For the below example, this means that you must change the base URI from `http://message-generation/greeting` to `https://message-generation/greeting`.

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 the Application section above), you have two options:

1. Use the `https://` URI scheme to access the service.
2. In your application configuration, set the `ribbon.IsSecure` property to `false`, and use the `http://` URI scheme to access the service.

If you wish to use HTTP rather than HTTPS, set `ribbon.IsSecure` first. In `application.yml`:

```
ribbon:
  IsSecure: false
```

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);
}
```

Greeter's `Greeting` class uses Jackson's `@JsonCreator` and `@JsonProperty` to read in the JSON response from Message Generation.

```
private static class Greeting {

    private String message;

    @JsonCreator
    public Greeting(@JsonProperty("message") String message) {
        this.message = message;
    }

    public String getMessage() {
        return this.message;
    }

}
```

Its `greeting()` method is mapped to the `/hello` endpoint and calls the `greeting()` method on `MessageGenerationClient` to get a greeting.

```
@RequestMapping(value = "/hello", method = GET)
public String greeting(@RequestParam(value="salutation", defaultValue="Hello") String salutation, @RequestParam(value="name", defaultValue="Bob") String name) {
  Greeting greeting =  messageGeneration.greeting(name, salutation);
  return greeting.getMessage();
}
```

Greeter now responds with a customizable `Greeting` when you access its `/hello` endpoint.

```
$ curl http://greeter.wise.com/hello
Hello, Bob!

$ curl http://greeter.wise.com/hello?name=John
Hello, John!
```

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 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):

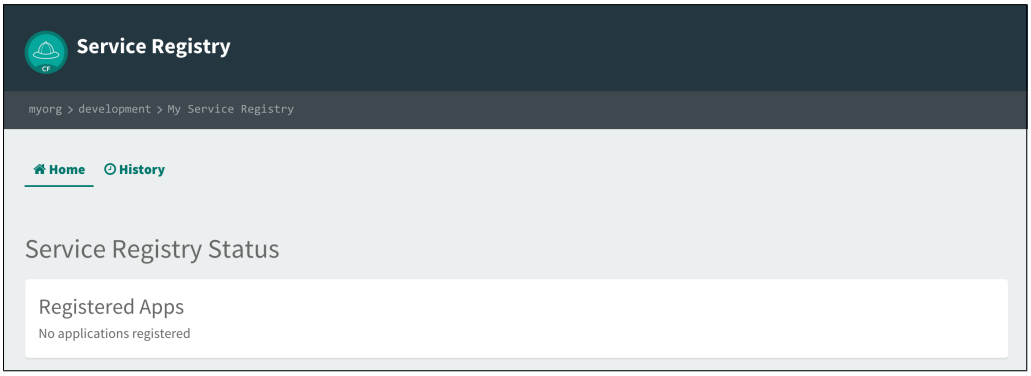
```
---

spring:
  profiles: dev

security:
  basic:
    enabled: false
```

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

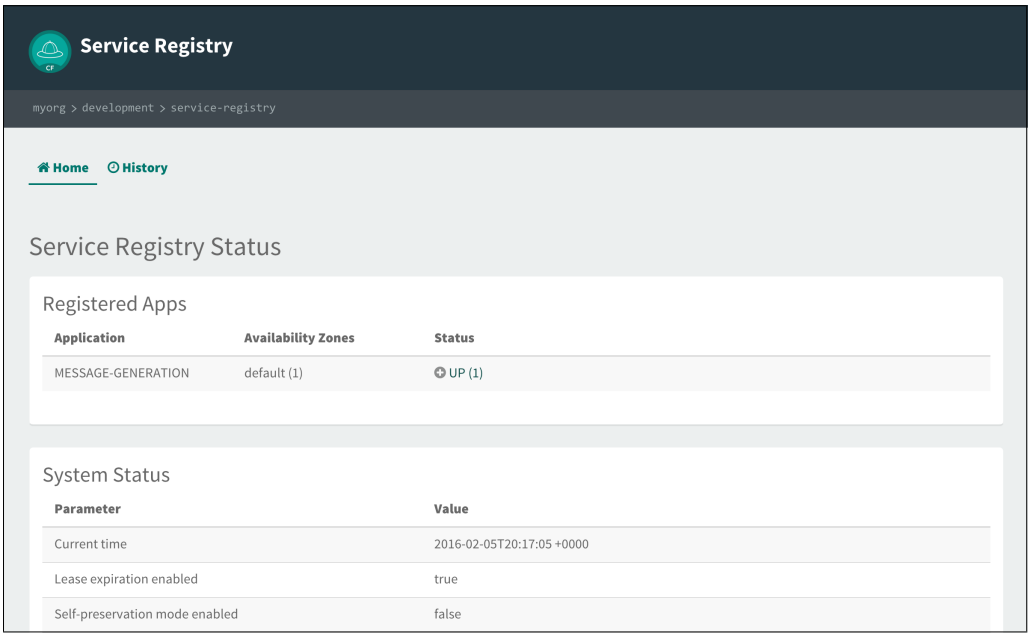
Using the Dashboard

Page last updated:

Before you have bound any applications to a Service Registry instance, the Service Registry dashboard will reflect that there are “No applications registered”.



To see applications listed in the dashboard, bind an application which uses `@EnableDiscoveryClient` to the service instance (see the Writing Client Applications subtopic). Once you have restaged the application and it has registered with Eureka, the dashboard will display it under **Registered Apps**.



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.

```
"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.apps.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 creates a Spring Cloud Netflix Eureka client configuration bean and registers it in the Spring application context. 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.

In a Spring application which is not using Spring Boot and will be bound to a Service Registry service instance, you can explicitly invoke the Spring Service Connector's service scanning to cause the Eureka client configuration to be created and registered. See "Scanning for Services" in the Spring Cloud Spring Service Connector documentation.

If you do not want to use either Spring Boot auto-configuration or service scanning, or if you want to further customize the client configuration in Java code, you can explicitly invoke the Spring Cloud Connectors configuration for Service Registry:

```
@Configuration
public class CloudConfig extends CloudConnectorsConfig {
 @Bean
 public EurekaClientConfigBean eurekaClientConfig() {
  return connectionFactory().eurekaClientConfig();
 }
}
```

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

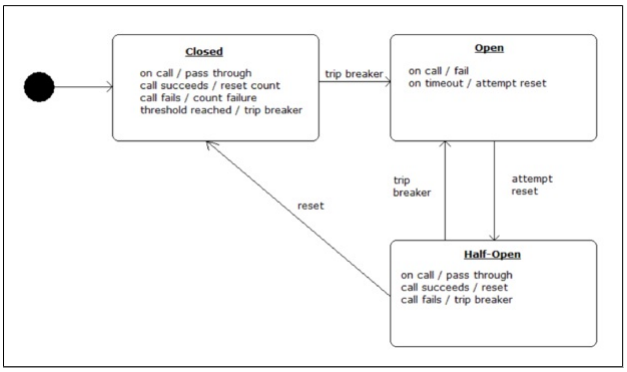
Page last updated:

- 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 for Pivotal Cloud Foundry®

Overview

Circuit Breaker Dashboard for Pivotal Cloud Foundry® (PCF) 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 for Pivotal Cloud Foundry® 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 topic.

Creating an Instance

Page last updated:

You can create a Circuit Breaker Dashboard instance using either the cf Command Line Interface tool 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 user...
OK

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

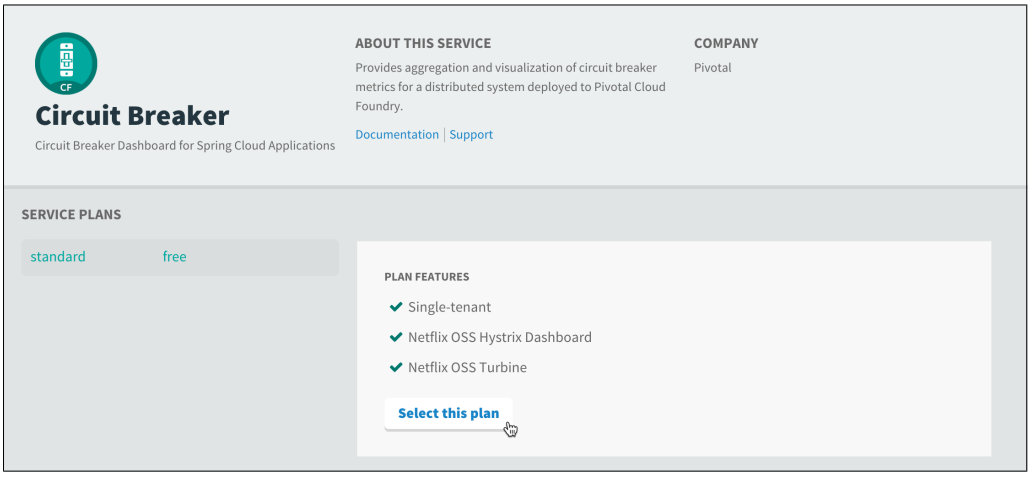
Run `cf create-service`, specifying the service, plan name, and instance name.

```
$ cf create-service p-circuit-breaker-dashboard standard My\ Circuit\ Breaker\ Dashboard
Creating service instance My Circuit Breaker Dashboard in org myorg / space development as user...
OK
```

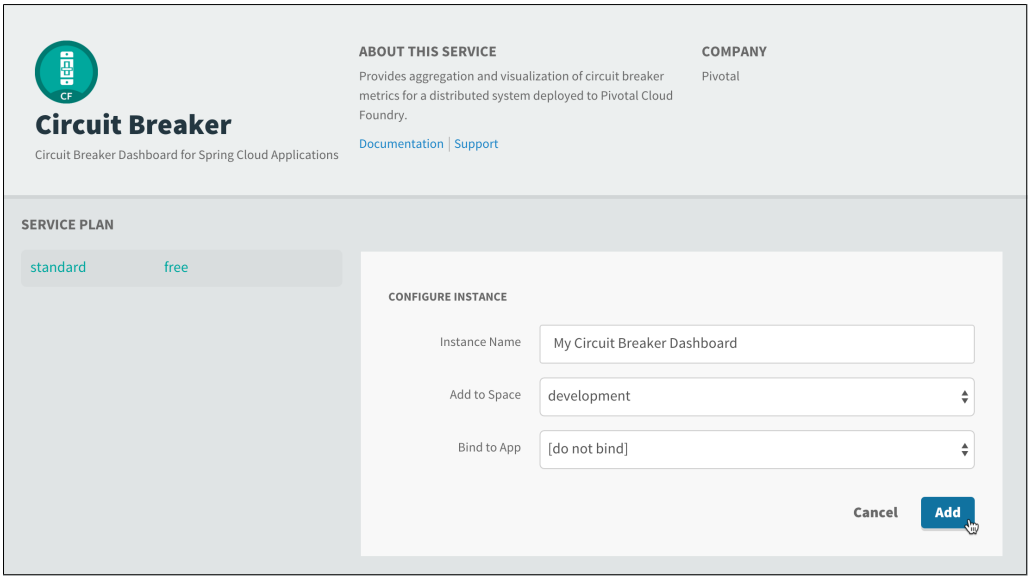
Using Apps Manager

Log into Apps Manager as a Space Developer. In the Marketplace, select **Circuit Breaker**.

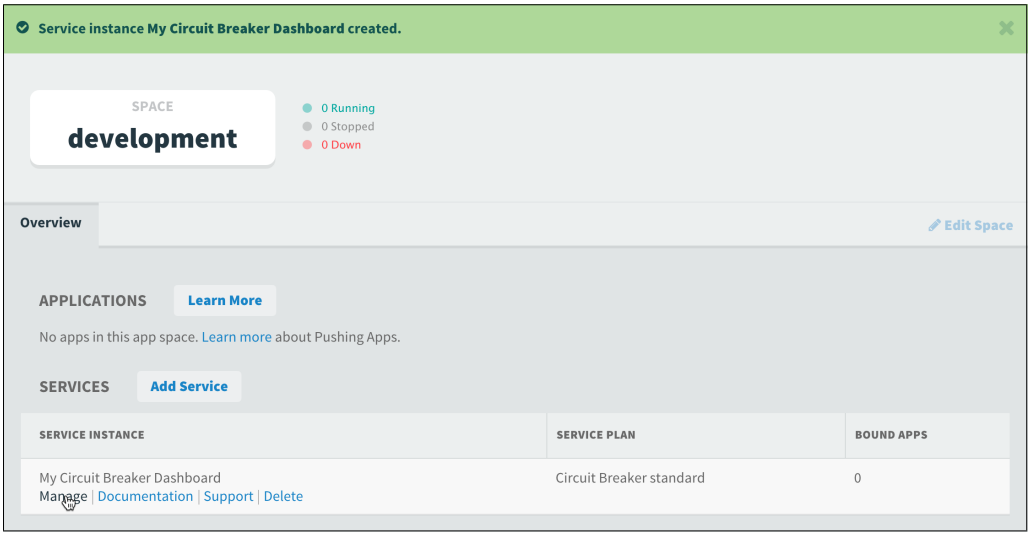
Select the desired plan for the new service.



Provide a name for the service (for example, “My Circuit Breaker Dashboard”). 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; 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 subtopic.

Writing Client Applications

Page last updated:

Refer to the sample applications in the "traveler" repository to follow along with the code in this subtopic.

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 subtopic.) 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:

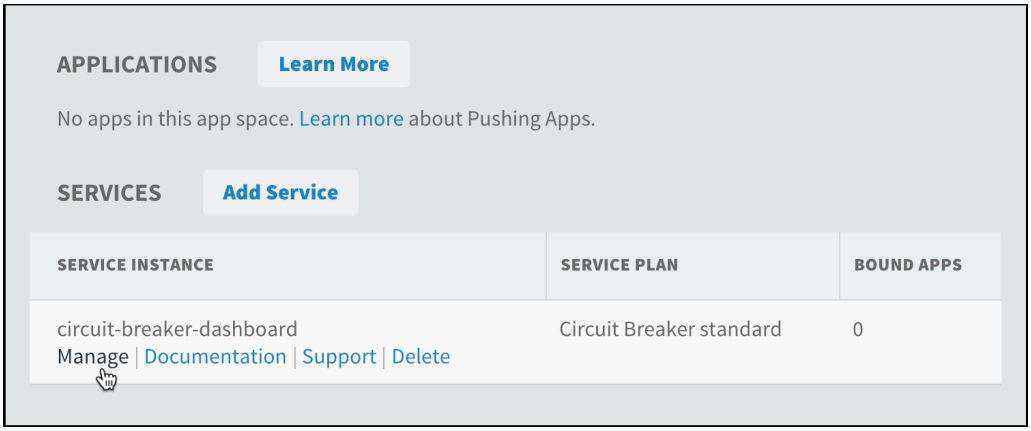
```
package agency;

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 Circuit Breaker Dashboard

Page last updated:

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



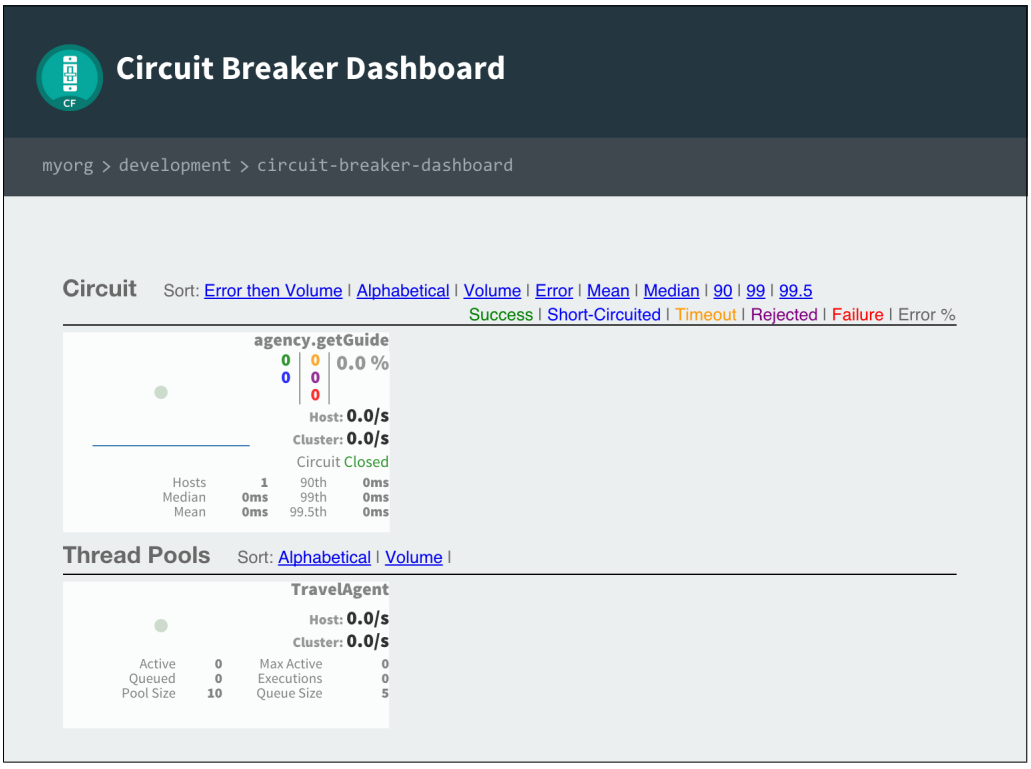
If you are using version 6.8.0 or later of the cf Command Line Interface Tool, you can also use `cf service SERVICE_NAME`, where `SERVICE_NAME` is the name of the Circuit Breaker Dashboard service instance:

```
$ cf service circuit-breaker-dashboard

Service instance: circuit-breaker-dashboard
Service: p-circuit-breaker-dashboard
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/e099ae0c-fafb-4d77-950f-5f87f2ca387f

Last Operation
Status: create succeeded
Message:
Started: 2016-05-17T03:51:48Z
Updated:
```

Visit the URL given for "Dashboard".

To see breaker statuses on the dashboard, configure an application as described in the Writing Client Applications subtopic, 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:

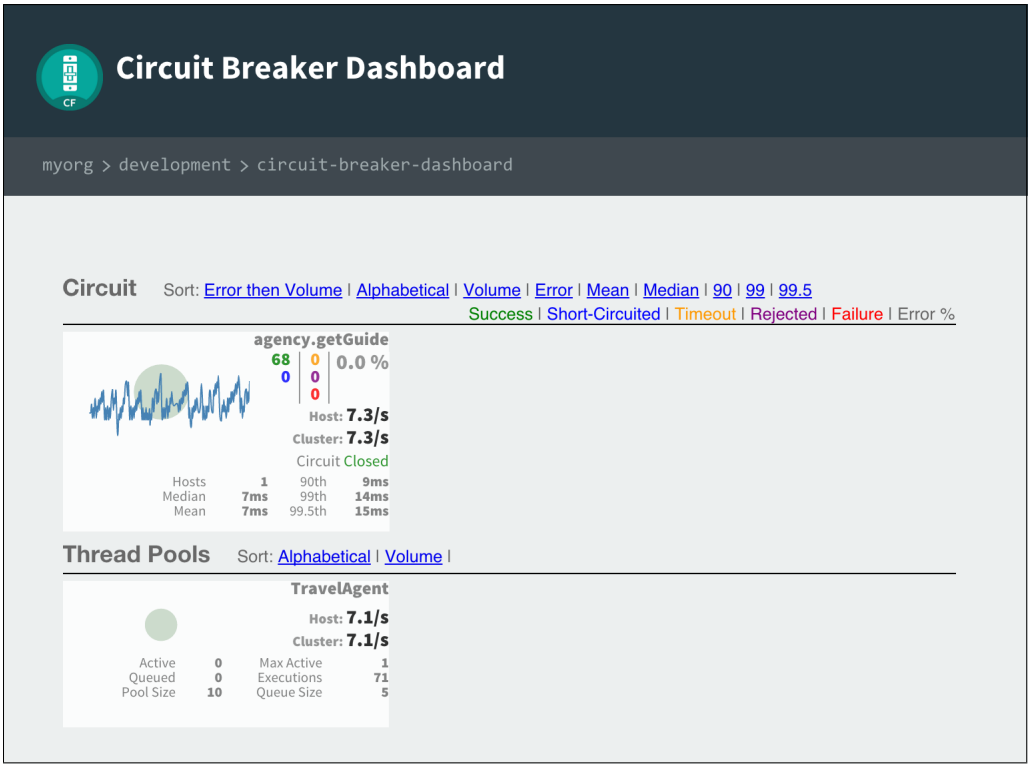


To see the circuit breaker in action, use curl, JMeter, Apache Bench, or similar to simulate load.

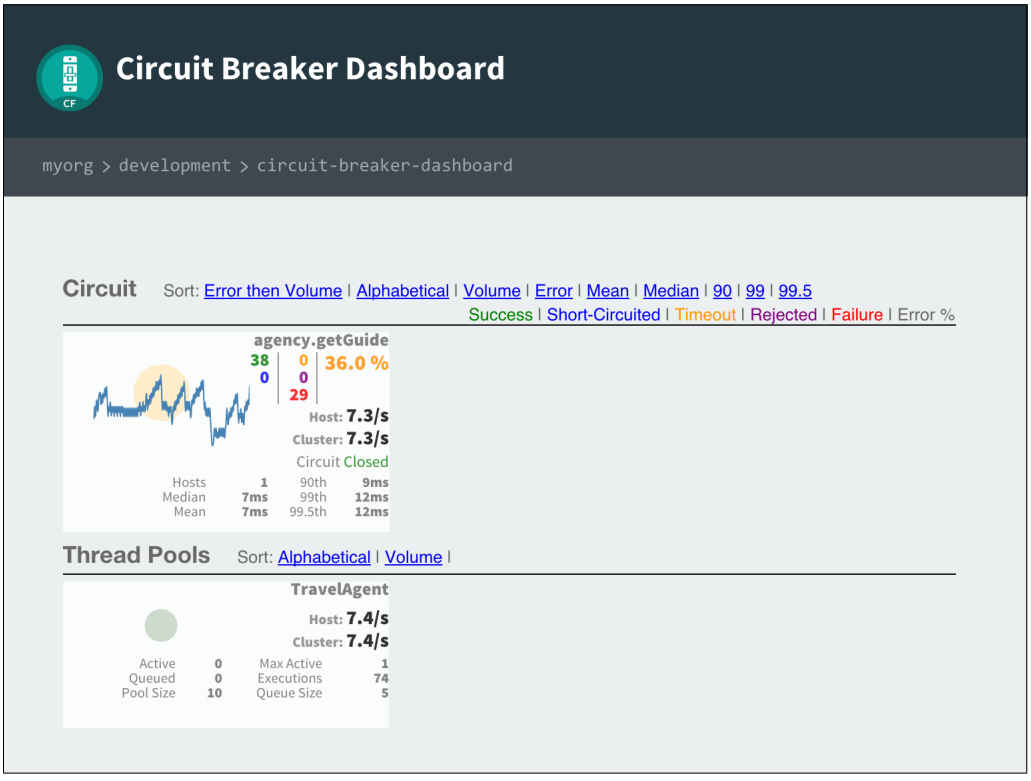
```
$ while true; do curl agency.wise.com; done
```

With the Company application running and available via the Service Registry instance (see the Writing Client Applications subtopic), 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.

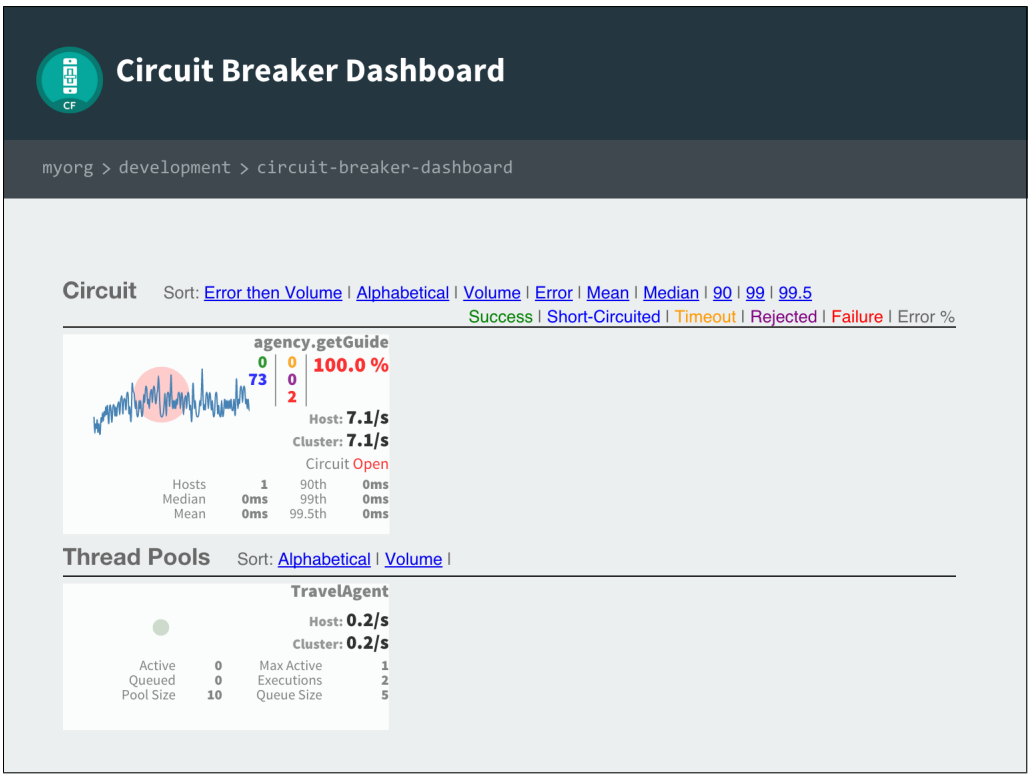
When service calls are succeeding, the circuit is closed, and the dashboard graph shows the rate of calls per second and successful calls per 10 seconds.



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 of a `VCAP_SERVICES` entry for the Spring Cloud Services Circuit Breaker Dashboard (edited for brevity).

```
"p-circuit-breaker-dashboard": [
 {
  "credentials": {
   "dashboard": {
    "uri": "https://hystrix-fd3842f2-ad07-40c0-9908-a763061f332c.apps.wise.com"
   },
   "stream": {
    "uri": "https://turbine-fd3842f2-ad07-40c0-9908-a763061f332c.apps.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`

Application Configuration

In a Spring Boot application which is bound to a Circuit Breaker Dashboard service instance, the connector automatically creates a Spring RabbitMQ `ConnectionFactory` bean using the credentials from the service binding. Spring Cloud Netflix Hystrix AMQP uses this `ConnectionFactory` to send metrics from the client application to a Netflix Turbine application, which aggregates metrics from all applications bound to the Circuit Breaker Dashboard service instance for display on the Hystrix dashboard.

In a Spring application which is not using Spring Boot and will be bound to a Circuit Breaker Dashboard service instance, you can explicitly invoke the Spring Service Connector's service scanning to cause the Hystrix AMQP `ConnectionFactory` to be created and registered. See "Scanning for Services" in the Spring Cloud Spring Service Connector documentation.

If you do not want to use either Spring Boot autoconfiguration or service scanning, or if you want to further customize the `ConnectionFactory` in Java code, you can explicitly invoke the Spring Cloud Connectors configuration for Circuit Breaker Dashboard:

```
@Configuration
public class CloudConfig extends CloudConnectorsConfig {
  @Bean
  public ConnectionFactory hystrixConnectionFactory() {
    return connectionFactory().hystrixConnectionFactory();
  }
}
```

Configure Multiple RabbitMQ Connections

If your client application uses RabbitMQ for its own business purposes and is bound to a Pivotal Cloud Foundry® (PCF) RabbitMQ service instance as well as to a Spring Cloud Services Circuit Breaker Dashboard service instance, then the client application may need two Spring RabbitMQ `ConnectionFactory` beans. In this scenario, Spring Cloud needs to know which `ConnectionFactory` bean to use for Hystrix AMQP.

To facilitate this, use explicit Spring Cloud Connectors configuration to create the two `ConnectionFactory` beans, using annotations to make the purpose of each bean clear.

One bean uses the `@HystrixConnectionFactory` annotation so that its created `ConnectionFactory` will be used by Hystrix AMQP:

```
@Configuration
public class CloudConfig extends CloudConnectorsConfig {
  @Bean
  @HystrixConnectionFactory
  public ConnectionFactory hystrixConnectionFactory() {
    return connectionFactory().hystrixConnectionFactory();
  }
}
```

Another `ConnectionFactory` bean—in this example, created in a separate configuration class—is marked with the `@Primary` annotation so that this `ConnectionFactory` will be used by your client application code:

```
@Configuration
public class RabbitConfig extends AbstractCloudConfig {
  @Bean
  @Primary
  public ConnectionFactory rabbitConnectionFactory() {
    return connectionFactory().rabbitConnectionFactory();
  }

  @Bean
  public DirectExchange exchange() {
    return new DirectExchange("sample-exchange", true, false);
  }

  @Bean
  public Queue queue() {
    return new Queue("sample-queue");
  }

  @Bean
  public Binding sampleBinding() {
    Queue queue = queue();
    DirectExchange exchange = exchange();
    return BindingBuilder.bind(queue).to(exchange).with(queue.getName());
  }
}
```

If you do use any explicit Spring Cloud Connectors configuration, it will disable Spring Boot's CloudAutoConfiguration, so you will need to explicitly declare beans for any other service to which your application is bound. For example, to explicitly declare a bean for a Netflix Eureka service:

```
@Bean
public EurekaClientConfigBean eurekaClientConfig() {
   return connectionFactory().eurekaClientConfig();
}
```

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 for Pivotal Cloud Foundry®)
- Home · Netflix/Hystrix Wiki (documentation for Netflix Hystrix, the library that underlies Circuit Breaker Dashboard for Pivotal Cloud Foundry®)
- 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)