

Table of Contents

Table of Contents	1
Spring Cloud Services for Pivotal Cloud Foundry	2
Installing Spring Cloud Services for Pivotal Cloud Foundry	3
Prerequisites	3
Installation	3
Config Server for Pivotal Cloud Foundry	7
Creating an Instance	8
The Config Server	11
Configuration Clients	13
Writing a Spring Client	16
Spring Cloud Connectors	20
Additional Resources	21
Service Registry for Pivotal Cloud Foundry	22
Creating an Instance	23
Registering a Service	26
Consuming a Service from a Client Application	28
Using the Dashboard	30
Spring Cloud Connectors	31
Additional Resources	32
Circuit Breaker Dashboard for Pivotal Cloud Foundry	33
Creating an Instance	34
Using a Circuit Breaker	37
Using the Circuit Breaker Dashboard	39
Spring Cloud Connectors	43
Additional Resources	44

Spring Cloud Services for Pivotal Cloud Foundry

Spring Cloud Services for Pivotal Cloud Foundry 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 starter POM in your application's dependencies and applying the appropriate annotation.

Services

Spring Cloud Services currently provides the following services:

- [Config Server](#)
- [Service Registry](#)
- [Circuit Breaker Dashboard](#)

Product Snapshot

Current Spring Cloud Services for PCF Details

Version: 0.1.1 BETA

Release Date: 27 July 2015

Software component version: Spring Cloud OSS Angel.SR2

Compatible Ops Manager Version(s): 1.5.x

Compatible Elastic Runtime Version(s): 1.5.x

vSphere support? Yes

AWS support? Yes

Installing Spring Cloud Services for Pivotal Cloud Foundry

NOTE: For the current BETA release of Spring Cloud Services, when installing alongside PCF Elastic Runtime 1.5.x, all experimental features which disable HTTP traffic must be unchecked.

Prerequisites

Spring Cloud Services requires the following Pivotal Cloud Foundry products:

- [MySQL for Pivotal Cloud Foundry](#)
- [RabbitMQ for Pivotal Cloud Foundry](#)

If they are not installed, you can follow the installation instructions below to install them along with Spring Cloud Services.

Spring Cloud Services also requires version 2.5 or greater of the Java buildpack to be the default Java buildpack on the Pivotal Cloud Foundry platform. You can use the cf Command Line Interface tool to see the version of the Java buildpack that is currently installed.

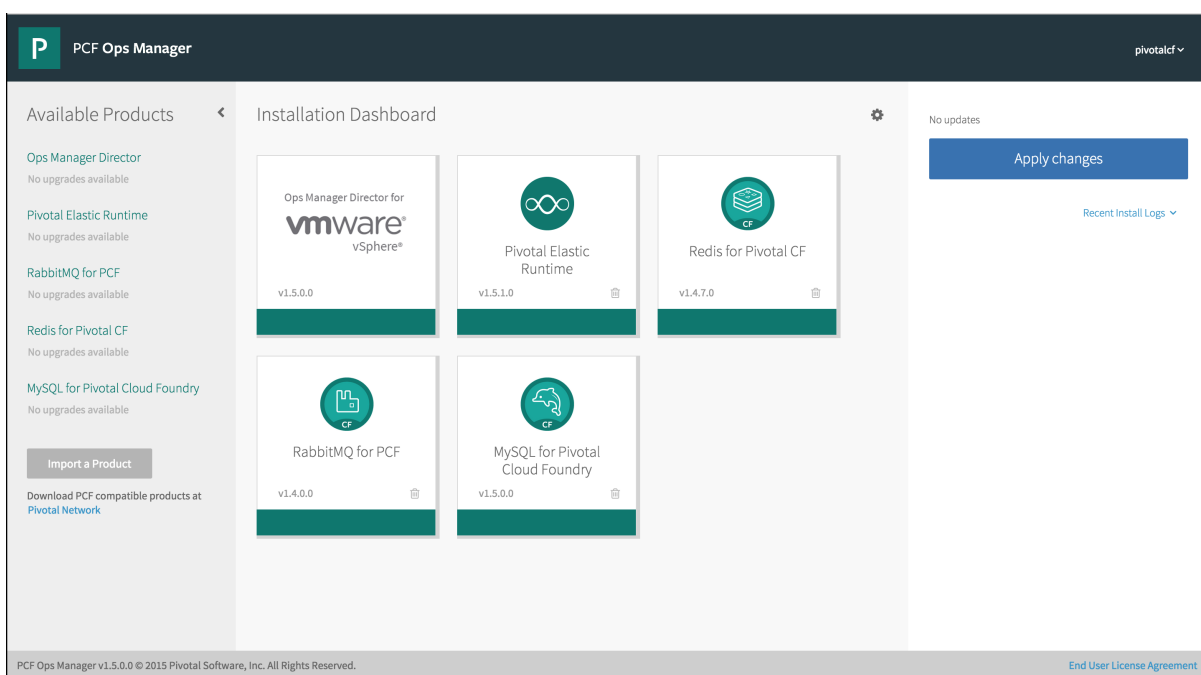
```
$ cf buildpacks
Getting buildpacks...

buildpack      position  enabled  locked  filename
java_buildpack_offline  1        true    false   java-buildpack-offline-v2.7.1.zip
ruby_buildpack    2        true    false   ruby_buildpack-cached-v1.3.0.zip
nodejs_buildpack  3        true    false   nodejs_buildpack-cached-v1.2.0.zip
```

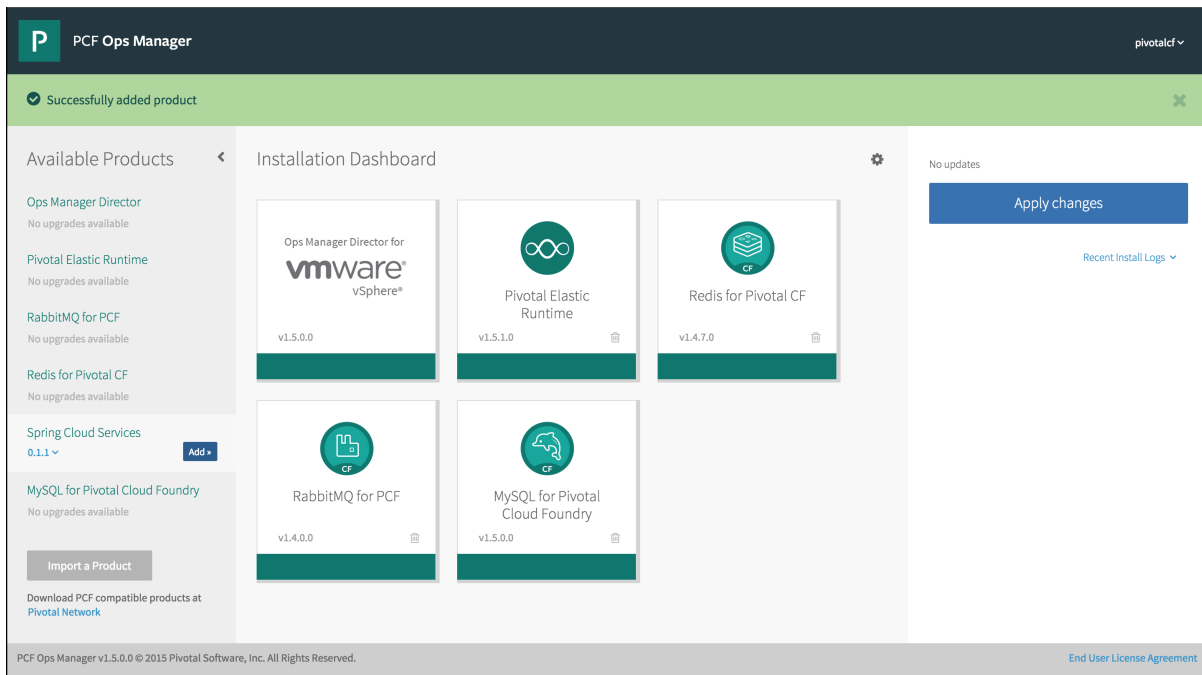
If the default Java buildpack is older than version 2.5, you can download a newer version from [Pivotal Network](#) and update Pivotal Cloud Foundry by following the instructions in the [Adding Buildpacks to Cloud Foundry](#) topic. If you do not delete or disable the older Java buildpack, make sure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

Installation

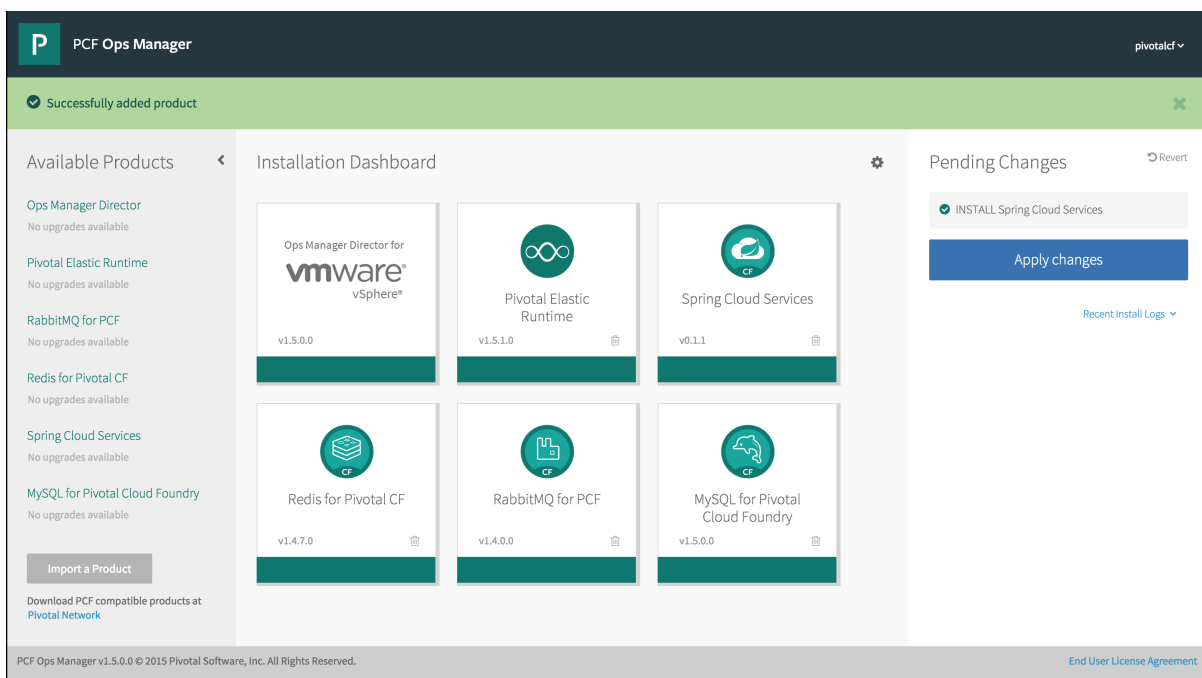
1. Download Spring Cloud Services from [Pivotal Network](#).
2. In Pivotal Cloud Foundry Operations Manager, click **Import a Product** on the left sidebar to upload the `p-spring-cloud-services-<version>.pivotal` file.



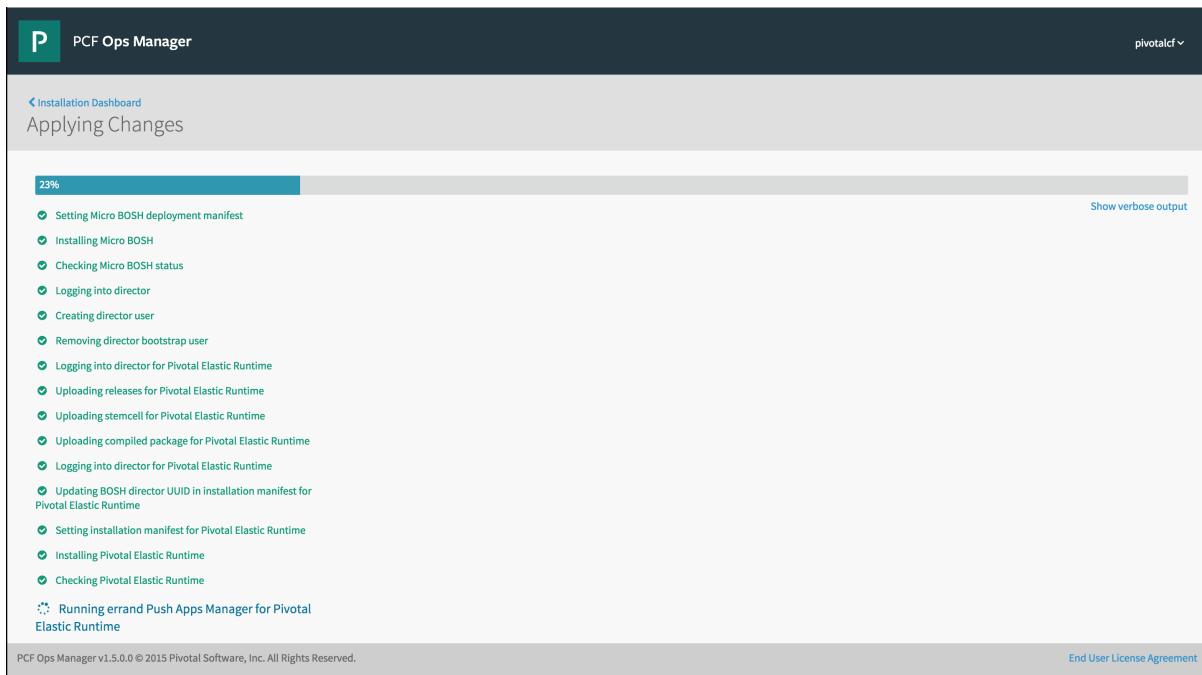
3. Hover over **Spring Cloud Services** in the **Available Products** list and click the **Add »** button.



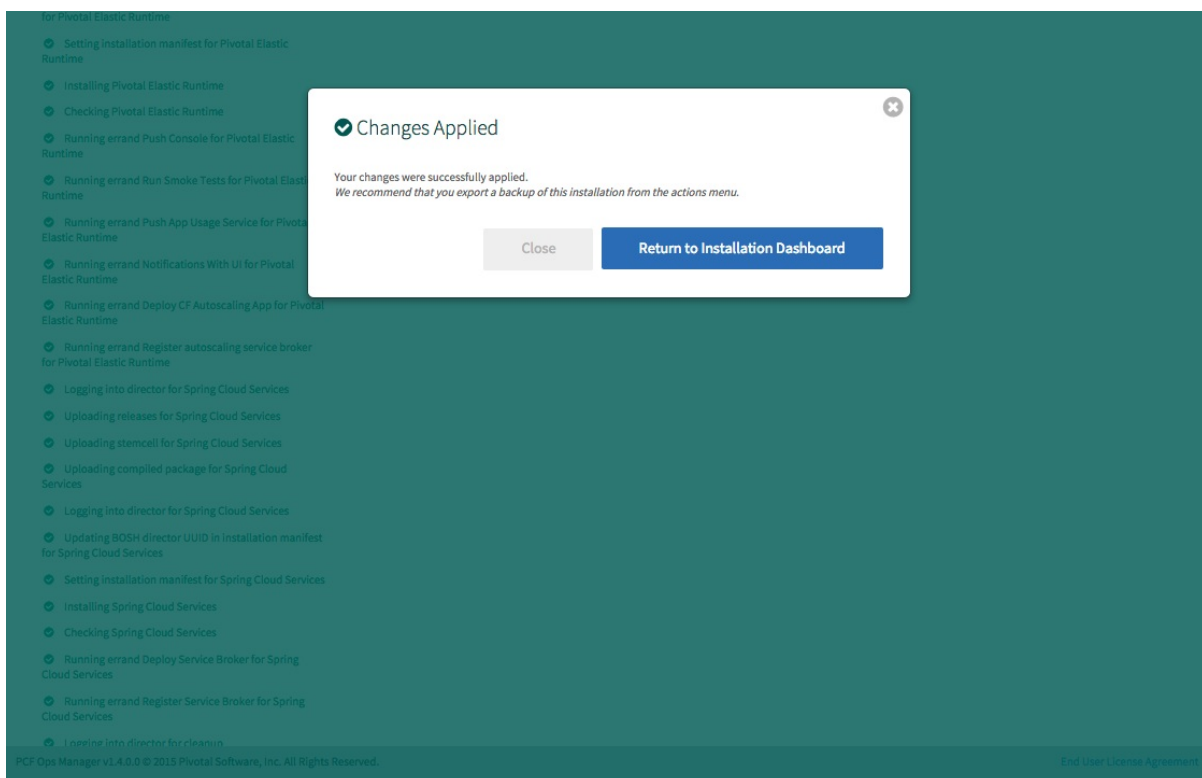
4. Start the installation process by clicking **Apply changes** on the right sidebar.



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



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



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

PCF Ops Manager pivotalcf

Available Products

- Ops Manager Director
No upgrades available
- Pivotal Elastic Runtime
No upgrades available
- RabbitMQ for PCF
No upgrades available
- Redis for Pivotal CF
No upgrades available
- Spring Cloud Services
No upgrades available
- MySQL for Pivotal Cloud Foundry
No upgrades available

[Import a Product](#)

Download PCF compatible products at [Pivotal Network](#)

Installation Dashboard

 Ops Manager Director for vmware vSphere® v1.5.0.0	 Pivotal Elastic Runtime v1.5.1.0	 Spring Cloud Services v0.1.1
 Redis for Pivotal CF v1.4.7.0	 RabbitMQ for PCF v1.4.0.0	 MySQL for Pivotal Cloud Foundry v1.5.0.0

No updates

[Apply changes](#)

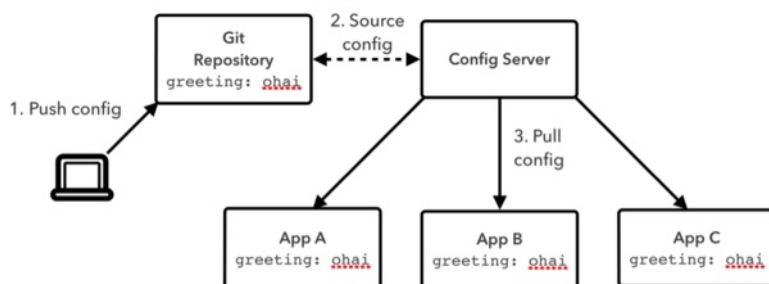
[Recent Install Logs](#)

PCF Ops Manager v1.5.0.0 © 2015 Pivotal Software, Inc. All Rights Reserved. [End User License Agreement](#)

Config Server for Pivotal Cloud Foundry

Overview

Config Server for Pivotal Cloud Foundry 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 also supports Subversion configuration.

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](#).

Please refer to the [“Cook” sample application](#) to follow along with code in this topic.

Creating an Instance

You can create a Config Server instance using either the cf Command Line Interface tool or Pivotal Cloud Foundry Apps Manager.

Using the cf CLI

Begin by targeting the correct space.

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

API endpoint: https://api.mydomain.com (API version: 2.25.0)
User: user
Org: myorg
Space: outer
```

If desired, view plan details for the Config Server product using `cf marketplace -s`.

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

service                plans      description
p-circuit-breaker-dashboard  standard  Circuit Breaker Dashboard for Spring Cloud Applications
p-config-server          standard  Config Server for Spring Cloud Applications
p-service-registry        standard  Service Registry for Spring Cloud Applications

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

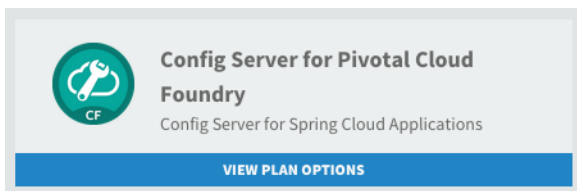
service plan  description  free or paid
standard      Standard Plan  free
```

Run `cf create-service`, specifying 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 outer as user...
OK
```

Using Apps Manager

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



Select the desired plan for the new service.



Config Server for Pivotal Cloud Foundry

Config Server for Spring Cloud Applications

ABOUT THIS SERVICE

Provides server and client-side support for externalized configuration in a distributed system deployed to Pivotal Cloud Foundry.

[Documentation](#) | [Support](#)

COMPANY

Pivotal

SERVICE PLANS


standardPrice unavailable

PLAN FEATURES

- ✓ Single-tenant
- ✓ Backed by user-provided Git repository

Select this plan

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



Config Server for Pivotal Cloud Foundry

Config Server for Spring Cloud Applications

ABOUT THIS SERVICE

Provides server and client-side support for externalized configuration in a distributed system deployed to Pivotal Cloud Foundry.

[Documentation](#) | [Support](#)

COMPANY

Pivotal

SERVICE PLAN

standardPrice unavailable

CONFIGURE INSTANCE

Instance Name

Add to Space

scs

⌵

Bind to App

[do not bind]

⌵

Cancel

Add

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

✓ Service instance My Config Server created.
✕

SPACE

development

[Edit Space](#)

APPLICATIONS

[Learn More](#)

STATUS	APP	INSTANCES	MEMORY
	hello hello.coral.springapps.io	1	512MB

SERVICES

[Add Service](#)

SERVICE INSTANCE	SERVICE PLAN	BOUND APPS
My Config Server Manage Documentation Support Delete	Config Server for Pivotal Cloud Foundry standard	0

Select a **Configuration Source** and 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. Hit the **Submit** button.

Spring Cloud Services for Pivotal Cloud Foundry

Config Server

Instance ID: 629d3303-4e91-452a-8060-b2724952e908

Configuration Source

☒ Git
☐ Subversion

Git URI

Git Branch (default is 'master')

Submit

Pivotal. © 2015 Pivotal Software Inc. All rights reserved.

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.

The Config Server

The Config Server serves configurations stored as either Java Properties files or YAML files. It reads files from a Git or Subversion 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`.

```
{
  "name": "myapp",
  "profiles": [
    "production"
  ],
  "label": "master",
  "propertySources": [
    {
      "name": "https://github.com/myorg/configurations/myapp-production.yml",
      "source": {
        "hello.message": "Hello Production!"
      }
    },
    {
      "name": "https://github.com/myorg/configurations/myapp.yml",
      "source": {
        "hello.message": "Hello Default!"
      }
    }
  ]
}
```

`{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**. For Subversion repositories, the default label is **trunk**. 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 (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

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

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

Bind a Service Instance to an Application

Using Apps Manager, select an application from the **Applications** list.

APPLICATIONS Learn More			
STATUS	APP	INSTANCES	MEMORY
	cook http://cookie.black.sp...	1	512MB

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

[Events](#)
[Services](#)
[Env Variables](#)
[Routes](#)
[Logs](#)
[Delete App](#)

BOUND SERVICES

[or add from Marketplace](#)

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 myapp config-server
Binding service config-server to app myapp in org myorg / space outer as user...
OK
```

Configuration Requests and Responses

After the 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 myapp
Getting env variables for app myapp in org myorg / space outer as ebenezer...
OK

System-Provided:
{
  "VCAP_SERVICES": {
    "p-config-server": [
      {
        "credentials": {
          "uri": "http://user:password@configserver.mydomain.com"
        },
        "label": "p-config-server",
        "name": "config-server",
        "plan": "standard",
        "tags": [
          "configuration",
          "spring-cloud"
        ]
      }
    ]
  }
}
```

The following is an example of making a request against the example Git repository outlined in the [The Config Server](#) subtopic, using the `uri` in the myapp Config Server's `credentials` object.

```
$ curl http://user:password@configserver.mydomain.com/myapp/production

{
  "name": "myapp",
  "profiles": [
    "production"
  ],
  "label": "master",
  "propertySources": [
    {
      "name": "https://github.com/myorg/configurations/myapp-production.yml",
      "source": {
        "hello.message": "Hello Production!"
      }
    },
    {
      "name": "https://github.com/myorg/configurations/myapp.yml",
      "source": {
        "hello.message": "Hello Default!"
      }
    }
  ]
}
```

In this example, the Server's response contains multiple values for the `hello.message` property. The client application must decide how to interpret such a response. The intent is that the first value in the list should take precedence over the others, so that in the above example, the client application should use "Hello Production!" as the value for `hello.message`. 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 "Request Paths" in 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 myapp SPRING_PROFILES_ACTIVE production
```

In the example `curl` request and response above, 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/myapp-production.yml` would have precedence over properties from `https://github.com/myorg/configurations/myapp.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 a Spring Client](#) subtopic.

Writing a Spring Client

Please refer to the [“Cook” sample application](#) to follow along with the code in this subtopic.

Follow the below instructions to use a Spring Boot application as a client for a Config Server instance.

Declare Dependencies

To use your Spring application as a Config Server client, you must include the `spring-cloud-services-starter-parent` BOM and declare `spring-cloud-services-starter-config-client` as a dependency.

If using Maven, [include in pom.xml](#):

```
<parent>
  <groupId>io.pivotal.spring.cloud</groupId>
  <artifactId>spring-cloud-services-starter-parent</artifactId>
  <version>1.0.0.M1</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
...
<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](#):

```
dependencyManagement {
    imports {
        mavenBom "io.pivotal.spring.cloud:spring-cloud-services-starter-parent:1.0.0.M1"
    }
}

...

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

(This will require [use of the Gradle dependency management plugin](#).)

You will also need to add `repo.spring.io/libs-snapshot` as a repository. [Using Maven](#):

```
<repository>
  <id>spring-snapshot</id>
  <url>https://repo.spring.io/libs-snapshot</url>
  <snapshots>
    <enabled>true</enabled>
  </snapshots>
</repository>
```

[Using Gradle](#):

```
repositories {
    ...
    maven {
        url "https://repo.spring.io/libs-snapshot"
    }
    ...
}
```

The dependencies bring in the Spring Cloud Config Server client library as well as the Spring Cloud Connectors Cloud Foundry Connector and Spring Cloud Services Connector. See the [Spring Cloud Connectors](#) subtopic for more information.

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 `application.properties` or `application.yml`.

In `application.yml` [↗](#):

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

This application will retrieve configuration properties from files named `cook*` in the Config Server's configuration source. You can see what configuration(s) the server will return by visiting the URI contained in the application's `VCAP_SERVICES` environment variable, appending the application name and a profile name (e.g., for the `default` profile, `/cook/default`).

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

System-Provided:
{
  "VCAP_SERVICES": {
    "p-config-server": [
      {
        "credentials": {
          "uri": "http://user:password@configserver.mydomain.com"
        },
        "label": "p-config-server",
        "name": "config-server",
        "plan": "standard",
        "tags": [
          "configuration",
          "spring-cloud"
        ]
      }
    ]
  }
}
...

$ curl https://user:password@configserver.mydomain.com/cook/default
{
  "name": "cook",
  "profiles": [
    "default"
  ],
  "label": "master",
  "propertySources": [
    {
      "name": "https://github.com/spring-cloud-samples/cook-config/cook.properties",
      "source": {
        "cook.special": "Pickled Cactus"
      }
    }
  ]
}
```

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 `.yaml` or `.properties` files in the Config Server instance's configuration source (the Git or Subversion repository). Filenames follow the format `{application}-{profile}.{extension}`, as in `cook-production.yaml`. (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.yaml](#).

```
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.yaml` above, the application will make a request of the Config Server using the path `/cook/production`, and the Config Server will return both `cook-production.properties` (the profile-specific configuration) and `cook.properties` (the default configuration).

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

Refresh Application Configuration

[Spring Boot Actuator](#) adds a `/refresh` endpoint to the application. A POST request to this endpoint will refresh any

`@RefreshScope` beans. You can use `@RefreshScope` to refresh properties which were initialized with values provided by the Config Server.

To begin, 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")
```

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: Fried Salamander

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

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

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

Spring Cloud Connectors

To connect client applications to the Config Server, Spring Cloud Services uses 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": {
      "uri": "http://4iCr7l:69e26t@config-1dc445ef-c09f.wise.com"
    },
    "label": "p-config-server",
    "name": "config-server",
    "plan": "standard",
    "tags": [
      "configuration",
      "spring-cloud"
    ]
  }
]
```

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

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

Config Server Detection Criteria

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

- `tags` including `configuration`
- `label` beginning with the `configuration` tag

See Also

For more information about the Spring Cloud Cloud Foundry Connector, see the following:

- [Spring Cloud Cloud Foundry Connector documentation](#)
- [Spring Cloud Connectors documentation](#)

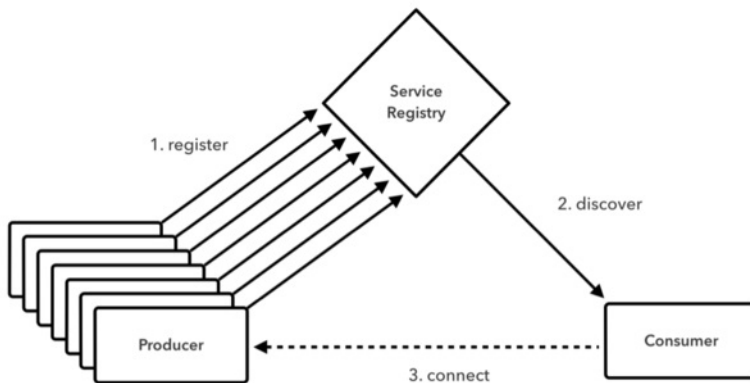
Additional Resources

- [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)
- [Spring Framework 3.1 M1 released](#) [↗](#) (release announcement which introduced Spring’s `Environment` abstraction and profiles)

Service Registry for Pivotal Cloud Foundry

Overview

Service Registry for Pivotal Cloud Foundry 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

You can create a Service Registry instance using either the cf Command Line Interface tool or Pivotal Cloud Foundry Apps Manager.

Using the cf CLI

Begin by targeting the correct space.

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

API endpoint: https://api.mydomain.com (API version: 2.25.0)
User: user
Org: myorg
Space: outer
```

If desired, view plan details for the Config Server product using `cf marketplace -s`.

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

service                plans                description
p-circuit-breaker-dashboard  standard            Circuit Breaker Dashboard for Spring Cloud Applications
p-config-server            standard            Config Server for Spring Cloud Applications
p-service-registry         standard            Service Registry for Spring Cloud Applications


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

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

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

Using Apps Manager

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




Service Registry for Pivotal Cloud Foundry

Service Registry for Spring Cloud Applications

VIEW PLAN OPTIONS

Select the desired plan for the new service.



Service Registry for Pivotal Cloud Foundry

Service Registry for Spring Cloud Applications

ABOUT THIS SERVICE

Provides application service registration and discovery in a distributed system deployed to Pivotal Cloud Foundry.

[Documentation](#) | [Support](#)

COMPANY

Pivotal

SERVICE PLANS

standard


Price unavailable

PLAN FEATURES

- ✓ Single-tenant
- ✓ Netflix OSS Eureka

Select this plan

Provide a name for the service (e.g., “service-registry”). Click the **Add** button.



Service Registry for Pivotal Cloud Foundry

Service Registry for Spring Cloud Applications

ABOUT THIS SERVICE

Provides application service registration and discovery in a distributed system deployed to Pivotal Cloud Foundry.

[Documentation](#) | [Support](#)

COMPANY

Pivotal

SERVICE PLAN

standard

Price unavailable

CONFIGURE INSTANCE

Instance Name

service-registry

Add to Space

scs

Bind to App

[do not bind]

Cancel

Add

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

✔ Service instance service-registry created.

SPACE
SCS

● 0 Running
● 0 Stopped
● 0 Down

Overview

[Edit Space](#)

APPLICATIONS

[Learn More](#)


No apps in this app space. [Learn more](#) about Pushing Apps.

SERVICES

[Add Service](#)

SERVICE INSTANCE	SERVICE PLAN	BOUND APPS
service-registry Manage Documentation Support Delete	Service Registry for Pivotal Cloud Foundry standard	0

It may take a few minutes to provision the service. Once it is ready, you will see a **Service Registry Dashboard** link. Click the link to enter the instance dashboard.



Spring Cloud Services for Pivotal Cloud Foundry

The Service Registry instance is now ready to be used. For information about registering a service application, see the [Registering a Service](#) subtopic.

Registering a Service

Refer to the sample applications in the [“greeting” repository](#) to follow along with the code in this subtopic.

Follow the below instructions to register a Spring application with the Service Registry.

Declare Dependencies

To register with a Service Registry instance, a Spring application must include the `spring-cloud-services-starter-parent` BOM and declare `spring-cloud-services-starter-service-registry` as a dependency.

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

```
<parent>
  <groupId>io.pivotal.spring.cloud</groupId>
  <artifactId>spring-cloud-services-starter-parent</artifactId>
  <version>1.0.0.M1</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
...
<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`:

```
dependencyManagement {
    imports {
        mavenBom "io.pivotal.spring.cloud:spring-cloud-services-starter-parent:1.0.0.M1"
    }
}

...

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

(This will require [use of the Gradle dependency management plugin](#).)

You will also need to add `repo.spring.io/libs-snapshot` as a repository. [Using Maven](#):

```
<repositories>
  <repository>
    <id>spring-snapshots</id>
    <url>https://repo.spring.io/libs-snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
</repositories>
```

[Using Gradle](#):

```
repositories {
    ...
    maven {
        url "https://repo.spring.io/libs-snapshot"
    }
}
```

The dependencies bring in the Spring Cloud Netflix client library for [Eureka](#), as well as the Spring Cloud Connectors Cloud Foundry Connector and Spring Cloud Services Connector. See the [Spring Cloud Connectors](#) subtopic for more information.

Identify the Application

Your service application must [include the `@EnableDiscoveryClient` annotation on a configuration class](#).

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

The `ExpeditionApplication` class also has a [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="camel boy") String name) {
    //...
    return new Greeting(salutation, name);
}
```


Set `spring.application.name` and `eureka.instance.hostname` to the values from the application deployment. You must also set `eureka.instance.nonSecurePort` for compatibility with Pivotal Cloud Foundry.

In [application.yml](#):

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

eureka:
  instance:
    hostname: ${vcap.application.uris[0]}
    nonSecurePort: 80
```

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).


Service Registry for Pivotal Cloud Foundry

[Home](#)
[History](#)

Service Registry Status

Registered Apps

Application	Availability Zones	Status
EXPEDITION	default (1)	UP (1)

For information about discovering and consuming registered services, see the [Consuming a Service](#) subtopic.

Consuming a Service from a Client Application

Refer to the sample applications in the [“greeting” repository](#) to follow along with the code in this subtopic.

Follow the below instructions to consume a service registered with the Service Registry from a Spring application.

Declare Dependencies

To consume a service which is registered with a Service Registry instance, a Spring application must include the `spring-cloud-services-starter-parent` BOM and declare `spring-cloud-services-starter-service-registry` as a dependency.

If using Maven, [include in](#) `pom.xml` [↗](#):

```
<parent>
  <groupId>io.pivotal.spring.cloud</groupId>
  <artifactId>spring-cloud-services-starter-parent</artifactId>
  <version>1.0.0.M1</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
...
<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` [↗](#):

```
dependencyManagement {
    imports {
        mavenBom "io.pivotal.spring.cloud:spring-cloud-services-starter-parent:1.0.0.M1"
    }
}
...

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

(This will require [use of the Gradle dependency management plugin](#) [↗](#).)

You will also need to add `repo.spring.io/libs-snapshot` as a repository. [Using Maven](#) [↗](#):

```
<repositories>
  <repository>
    <id>spring-snapshots</id>
    <url>https://repo.spring.io/libs-snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
</repositories>
```

[Using Gradle](#) [↗](#):

```
repositories {
    ...
    maven {
        url "https://repo.spring.io/libs-snapshot"
    }
}
```

The dependencies bring in the Spring Cloud Netflix client library for [Eureka](#) [↗](#), as well as the Spring Cloud Connectors Cloud Foundry Connector and Spring Cloud Services Connector. See the [Spring Cloud Connectors](#) subtopic for more information.

Discover and Consume a Service

A consuming application must [include the `@EnableDiscoveryClient` annotation on a configuration class](#).

```
@SpringBootApplication
@EnableDiscoveryClient
@RestController
public class CamelboyApplication {

    @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.

The Expedition application is registered with the Service Registry instance as **expedition**, so in the Camelboy application, [the `hello\(\)` method on the `CamelboyApplication` class](#) uses the base URI `http://expedition` to get a greeting message from Expedition.

```
@RequestMapping(value = "/hello", method = RequestMethod.GET)
public String hello(@RequestParam(value="salutation",
                                defaultValue="Hello") String salutation,
                  @RequestParam(value="name",
                                defaultValue="camel boy") String name) {
    URI uri = UriComponentsBuilder.fromUriString("http://expedition/greeting")
        .queryParam("salutation", salutation)
        .queryParam("name", name)
        .build()
        .toUri();

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

Camelboy's [Greeting class](#) uses Jackson's [@JsonCreator](#) and [@JsonProperty](#) to read in the JSON response from Expedition.

```
private static class Greeting {

    private String message;

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

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

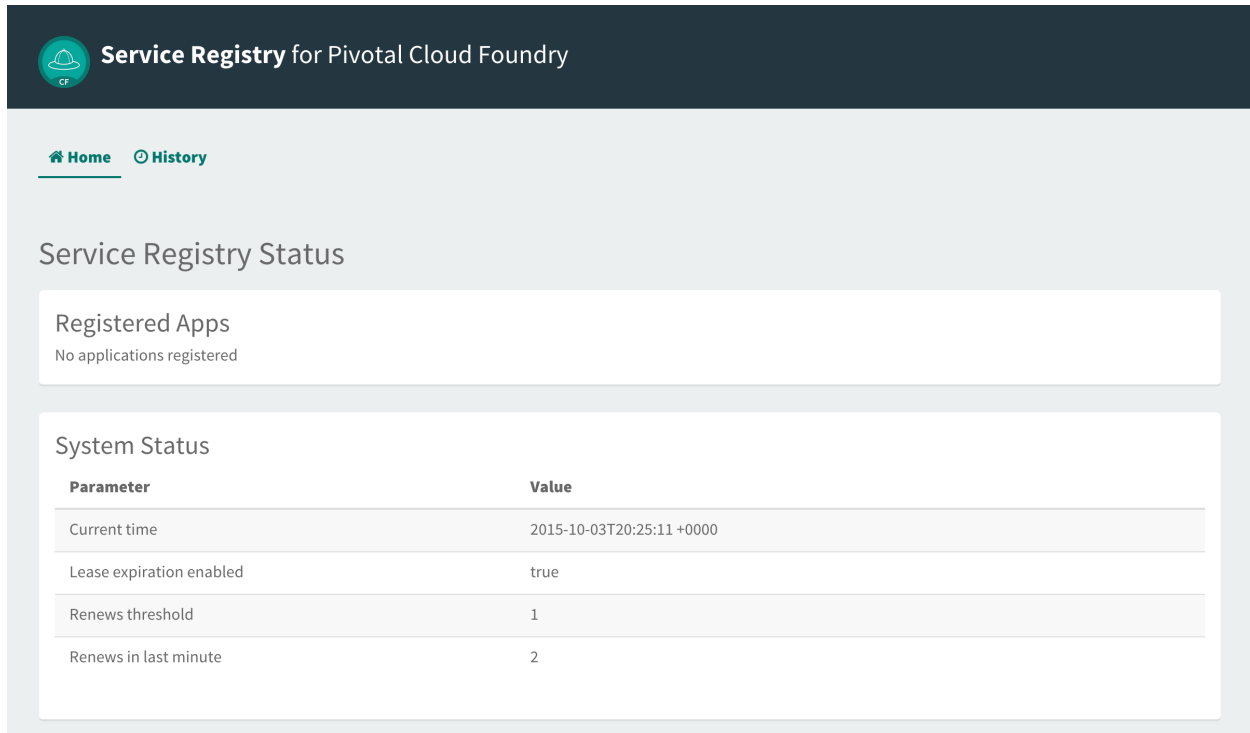
The Camelboy application now responds with a customizable `Greeting` when we access its `/hello` endpoint.

```
$ curl http://camelboy.wise.com/hello
Hello, camel boy!

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

Using the Dashboard

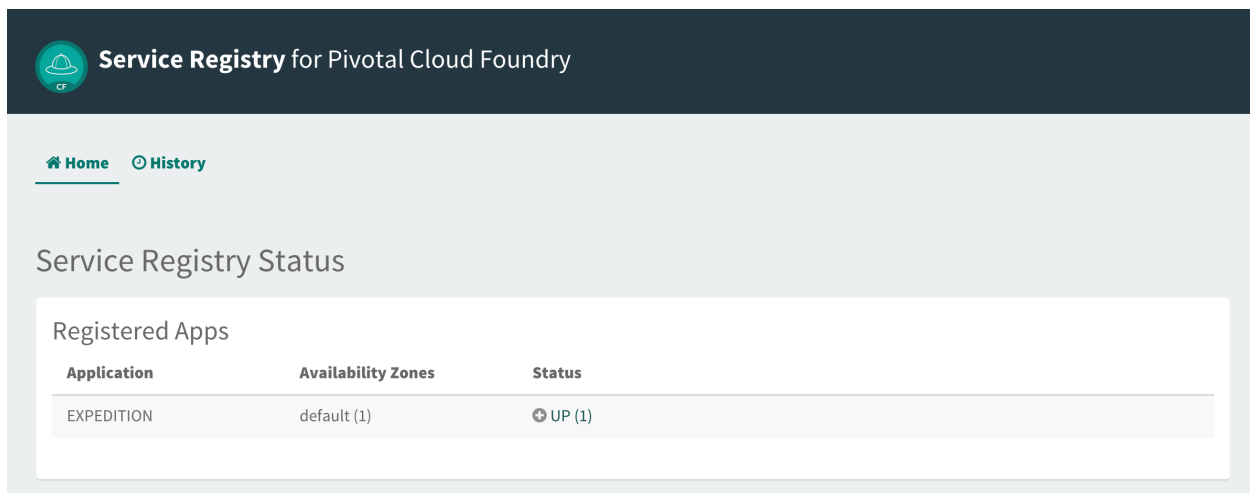
Before you have bound any applications to the Service Registry instance, the Service Registry dashboard will reflect that there are “No applications registered”.



The screenshot shows the 'Service Registry for Pivotal Cloud Foundry' dashboard. At the top, there's a dark header with the Pivotal logo and the title. Below the header, there are two tabs: 'Home' (selected) and 'History'. The main section is titled 'Service Registry Status'. Under this, there's a 'Registered Apps' section which displays 'No applications registered'. Below that, there's a 'System Status' section containing a table with system parameters and their values.

Parameter	Value
Current time	2015-10-03T20:25:11 +0000
Lease expiration enabled	true
Renews threshold	1
Renews in last minute	2

To see applications listed in the dashboard, bind an application which uses `@EnableDiscoveryClient` to the service instance (see the [Registering a Service](#) and [Consuming a Service](#) subtopics). Once you have restaged the application and it has registered with Eureka, the dashboard will display it under **Registered Apps**.



This screenshot shows the same dashboard but with one application registered. The 'Registered Apps' section now displays a table with the following data:

Application	Availability Zones	Status
EXPEDITION	default (1)	UP (1)

Spring Cloud Connectors

To connect client applications to the Service Registry, Spring Cloud Services uses 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": {
      "uri": "http://R5KYq8:0AyQ0x4y@eureka-8982c70c-9ad1.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`
- `label` beginning with the `eureka` tag

See Also

For more information about the Spring Cloud Cloud Foundry Connector, see the following:

- [Spring Cloud Cloud Foundry Connector documentation](#)
- [Spring Cloud Connectors documentation](#)

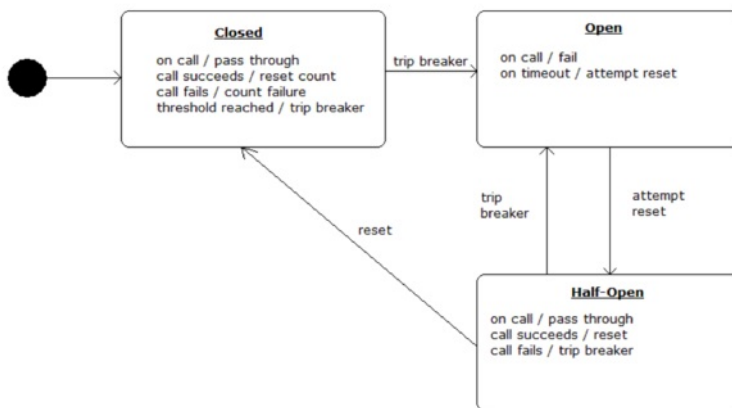
Additional Resources

- [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 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 monitor a service for failure, prevent failures from cascading, and supply dependent services until the failing service is operable again.



When applied to a service, a circuit breaker watches for failing calls to a 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

You can create a Circuit Breaker Dashboard instance using either the cf Command Line Interface tool or Pivotal Cloud Foundry Apps Manager.

Using the cf CLI

Begin by targeting the correct space.

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

API endpoint:   https://api.mydomain.com (API version: 2.25.0)
User:          user
Org:           myorg
Space:         outer
```

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 outer as user...
OK

service          plans          description
p-circuit-breaker-dashboard  standard      Circuit Breaker Dashboard for Spring Cloud Applications
p-config-server             standard      Config Server for Spring Cloud Applications
p-service-registry          standard      Service Registry for Spring Cloud Applications

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

$ cf marketplace -s p-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 circuit-breaker-dashboard
Creating service instance circuit-breaker-dashboard in org myorg / space outer as user...
OK
```

Using Apps Manager

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




Circuit Breaker for Pivotal Cloud Foundry

Circuit Breaker Dashboard for Spring Cloud Applications

[VIEW PLAN OPTIONS](#)

Select the desired plan for the new service.



Circuit Breaker for Pivotal Cloud Foundry

Circuit Breaker Dashboard for Spring Cloud Applications

ABOUT THIS SERVICE

Provides aggregation and visualization of circuit breaker metrics for a distributed system deployed to Pivotal Cloud Foundry.

[Documentation](#) | [Support](#)

COMPANY

Pivotal

SERVICE PLANS

standard


Price unavailable

PLAN FEATURES

- ✓ Single-tenant
- ✓ Netflix OSS Hystrix Dashboard
- ✓ Netflix OSS Turbine

[Select this plan](#)

Provide a name for the service (e.g. “circuit-breaker-dashboard”). Click the **Add** button.



Circuit Breaker for Pivotal Cloud Foundry

Circuit Breaker Dashboard for Spring Cloud Applications

ABOUT THIS SERVICE

Provides aggregation and visualization of circuit breaker metrics for a distributed system deployed to Pivotal Cloud Foundry.

[Documentation](#) | [Support](#)

COMPANY

Pivotal

SERVICE PLAN

standard

Price unavailable

CONFIGURE INSTANCE

Instance Name

Add to Space

Bind to App

[Cancel](#) [Add](#)

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

Service instance circuit-breaker-dashboard created.

SPACE

SCS

0 Running

0 Stopped

0 Down

Overview

Edit Space

APPLICATIONS

Learn More


No apps in this app space. [Learn more](#) about Pushing Apps.

SERVICES

Add Service

SERVICE INSTANCE	SERVICE PLAN	BOUND APPS
circuit-breaker-dashboard Manage Documentation Support Delete	Circuit Breaker for Pivotal Cloud Foundry standard	0

It may take a few minutes to provision the dashboard. Once it is ready, you will see a **Circuit Breaker Dashboard** link. Click the link to enter the instance dashboard.

 **Spring Cloud Services** for Pivotal Cloud Foundry

Circuit Breaker

Instance ID: d1305fdb-44c4-498b-95f8-17d85a2815f8

[Circuit Breaker Dashboard](#)

Pivotal

© 2015 Pivotal Software Inc. All rights reserved.

The Circuit Breaker Dashboard instance is now ready to be used. For an example of using a circuit breaker in an application, see the [Using a Circuit Breaker](#) subtopic.

Using a Circuit Breaker

Please refer to the sample applications in the [“traveler” repository](#) to follow along with the code in this subtopic.

Follow the below instructions to use a circuit breaker in a Spring application.

Declare Dependencies

Your application must include the `spring-cloud-services-starter-parent` BOM and declare `spring-cloud-services-starter-circuit-breaker` as a dependency.

If using Maven, [include in pom.xml](#):

```
<parent>
  <groupId>io.pivotal.spring.cloud</groupId>
  <artifactId>spring-cloud-services-starter-parent</artifactId>
  <version>1.0.0.M1</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
...
<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](#):

```
dependencyManagement {
  imports {
    mavenBom "io.pivotal.spring.cloud:spring-cloud-services-starter-parent:1.0.0.M1"
  }
}
...
dependencies {
  ...
  compile("io.pivotal.spring.cloud:spring-cloud-services-starter-circuit-breaker")
}
```

(This will require [use of the Gradle dependency management plugin](#).)

You must also add `repo.spring.io/libs-snapshot` as a repository. [Using Maven](#):

```
<repositories>
  <repository>
    <id>spring-snapshot</id>
    <url>https://repo.spring.io/libs-snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
</repositories>
```

[Using Gradle](#):

```
repositories {
  ...
  maven {
    url "https://repo.spring.io/libs-snapshot"
  }
}
```

The dependencies bring in the Spring Cloud Netflix client library for [Hystrix](#), as well as the Spring Cloud Connectors Cloud Foundry Connector and Spring Cloud Services Connector. See the [Spring Cloud Connectors](#) subtopic for more information.

Apply the Circuit Breaker Pattern

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 [Consuming a Service](#) 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\(\)](#).

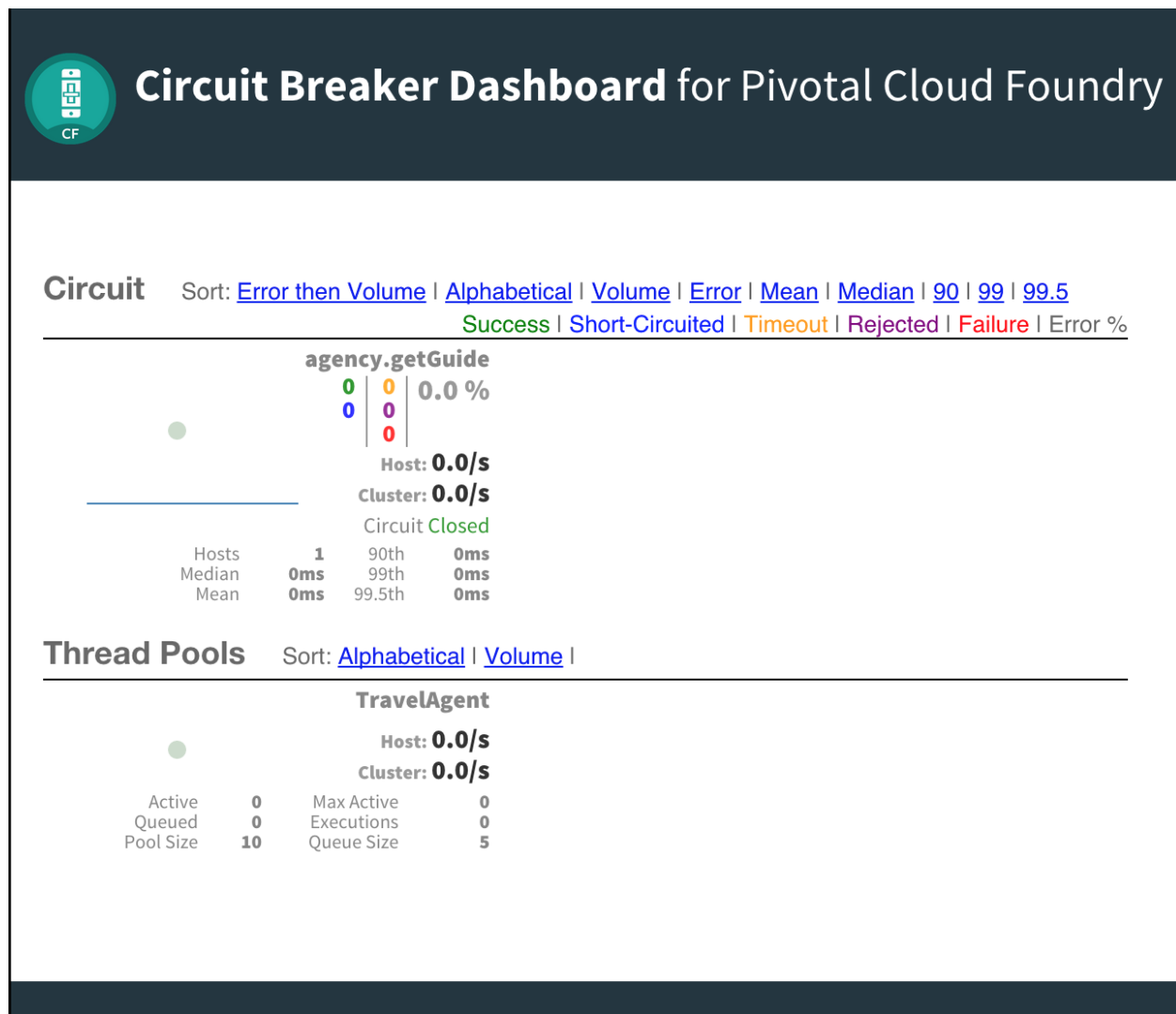
```
String getBackupGuide() {
    return "None available! Your backup guide is: Cookie";
}
```

If you wish, you can also annotate fallback methods themselves with `@HystrixCommand` to create a fallback chain.

Using the Circuit Breaker Dashboard

To see breaker statuses on the dashboard, configure an application as described in the [Using a Circuit Breaker](#) 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 restarted, 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 [Using a Circuit Breaker](#) 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.

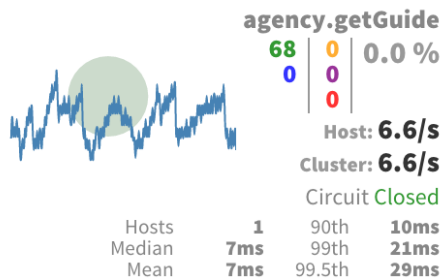


Circuit Breaker Dashboard for Pivotal Cloud Foundry

Circuit

Sort: [Error then Volume](#) | [Alphabetical](#) | [Volume](#) | [Error](#) | [Mean](#) | [Median](#) | [90](#) | [99](#) | [99.5](#)

[Success](#) | [Short-Circuited](#) | [Timeout](#) | [Rejected](#) | [Failure](#) | Error %



Thread Pools

Sort: [Alphabetical](#) | [Volume](#) |



When calls begin to fail, the graph shows the rate of failed calls in red.

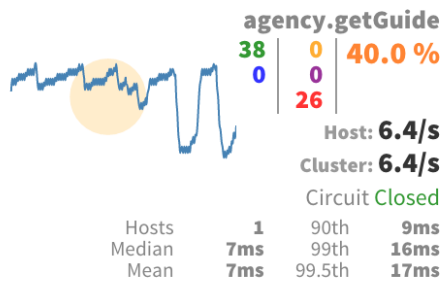


Circuit Breaker Dashboard for Pivotal Cloud Foundry

Circuit

Sort: [Error then Volume](#) | [Alphabetical](#) | [Volume](#) | [Error](#) | [Mean](#) | [Median](#) | [90](#) | [99](#) | [99.5](#)

[Success](#) | [Short-Circuited](#) | [Timeout](#) | [Rejected](#) | [Failure](#) | Error %



Thread Pools

Sort: [Alphabetical](#) | [Volume](#) |



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.

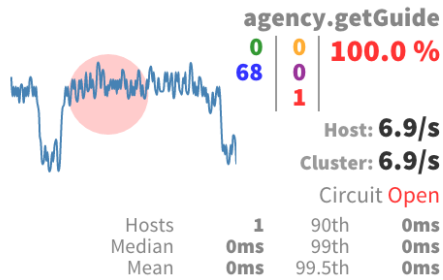


Circuit Breaker Dashboard for Pivotal Cloud Foundry

Circuit

Sort: [Error then Volume](#) | [Alphabetical](#) | [Volume](#) | [Error](#) | [Mean](#) | [Median](#) | [90](#) | [99](#) | [99.5](#)

[Success](#) | [Short-Circuited](#) | [Timeout](#) | [Rejected](#) | [Failure](#) | Error %



Thread Pools

Sort: [Alphabetical](#) | [Volume](#) |



With the circuit breaker in place on its `getGuide()` method, the Agency example application never returns a status code other than `200` to the requester.

Spring Cloud Connectors

To connect client applications to the Circuit Breaker Dashboard, Spring Cloud Services uses 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": "http://hystrix-99dae898-35ef-4073-9d46-ab24e9dee6f0.black.springapps.io"
      },
      "stream": {
        "uri": "http://turbine-99dae898-35ef-4073-9d46-ab24e9dee6f0.black.springapps.io"
      }
    },
    "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`
- `label` beginning with the `hystrix-amqp` tag

See Also

For more information about the Spring Cloud Cloud Foundry Connector, see the following:

- [Spring Cloud Cloud Foundry Connector documentation](#)
- [Spring Cloud Connectors documentation](#)

Additional Resources

- [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)