Pivotal Cloud Foundry

Overview

Pivotal Cloud Foundry®, powered by Cloud Foundry, delivers a turnkey PaaS experience on multiple infrastructures with leading application and data services.

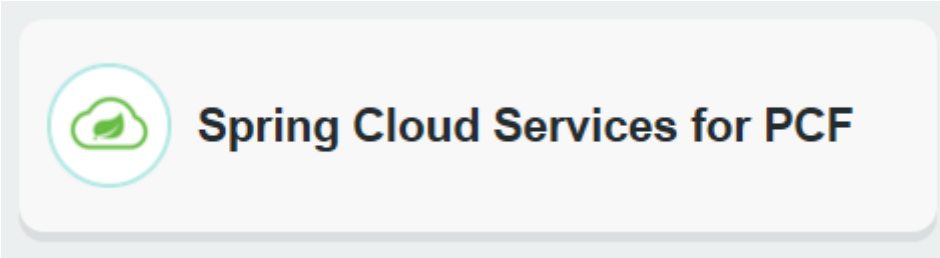
Features

- Commercially supported release based on Cloud Foundry open source
- Instant, horizontal application tier scaling
- Web console for resource management and administration of applications and services
- Applications benefit from built-in services like load balancing and DNS, automated health management, logging and auditing

- Java Spring support through provided Java buildpack
- Optimized developer experience for Spring framework
- MySQL Service for rapid development and testing
- Automatic application binding and service provisioning for Pivotal Services such as Pivotal RabbitMQ and MySQL for PCF.

- Pivotal Cloud Foundry Runtime for Windows
- Microsoft Azure Service Broker for PCF
- Pivotal Cloud Foundry Log Search
- Pivotal Cloud Foundry Service Broker for AWS
- Push Notification for PCF
- Single Sign-On for PCF
- Spring Cloud Services for PCF
- GCP Service Broker for PCF
- Pivotal Cloud Foundry Elastic Runtime
- Pivotal Cloud Foundry Metrics
- Pivotal tc Server
- Pivotal Cloud Foundry JMX Bridge (Ops Metrics)
- Pivotal Cloud Foundry Operations Manager
- IPsec Add-on for PCF
- Pivotal Cloud Foundry Isolation Segment

Spring Cloud Services for PCF



Spring Cloud (http://projects.spring.io/spring-cloud/) provides tools for Spring developers to quickly apply some of the common patterns found in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus).

Coordination of distributed systems leads to boiler plate patterns, and using Spring Cloud, developers can quickly stand up services and applications that implement those patterns.

The Spring Cloud Services suite adds several of the central coordination services found in Spring Cloud to the Pivotal Cloud Foundry Marketplace.

Features

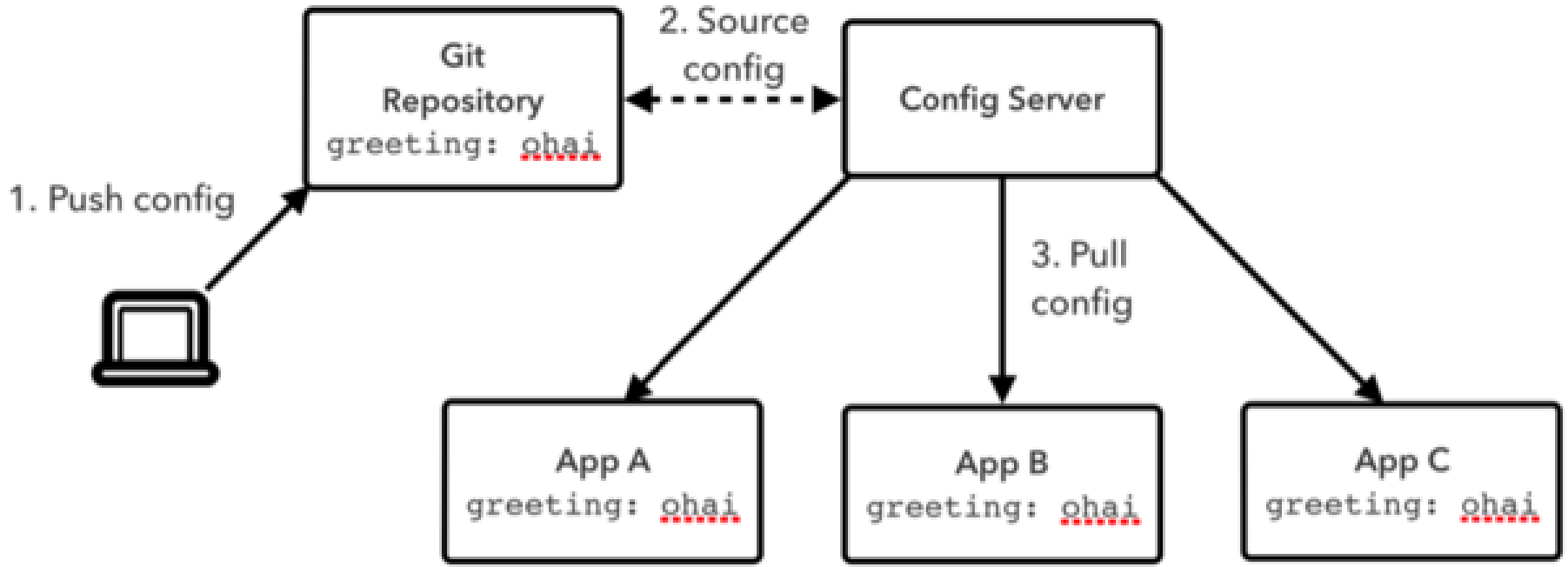
- ❑ Config Server (based on Spring Cloud Config Server)
- ❑ Service Registry (based on Eureka via Spring Cloud Netflix)
- ❑ Circuit Breaker Dashboard (based on Hystrix and Turbine via Spring Cloud Netflix)

Config Server for Pivotal Cloud Foundry

- Config Server for Pivotal Cloud Foundry (PCF) is an externalized application configuration service, which gives us 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, we can use Config Server to manage the configuration between environments and be certain that the application has everything it needs to run when we 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.


Git
Repository
greeting: ohai
2. Source
config
Config Server
1. Push config
3. Pull
config
App A
greeting: ohai
App B
greeting: ohai
App C
greeting: ohai

The concepts on both client and server map identically to the Spring Environment and PropertySource abstractions. They work very well with Spring applications, but can be applied to applications written in any language.

The default implementation of the server storage backend uses Git. HashiCorp Vault is also supported.

Config Server for Pivotal Cloud Foundry is based on Spring Cloud Config Server.

Creating an Instance

We can create a Config Server service instance using either

- ❑ The Cloud Foundry Command Line Interface tool (cf CLI) or
- ❑ Pivotal Cloud Foundry® Apps Manager

Using the cf CLI

Begin by targeting the correct org and space.

$ cf target -o scp -s development

api endpoint: https://api.run.pivotal.io
api version: 2.97.0
user: jbossramana@gmail.com
org: scp
space: development

We can view plan details for the Config Server product:

$ cf marketplace

Getting services from marketplace in org scp / 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

To create an instance, specifying settings for Git configuration sources :

cf create-service -c '{ "git": { "uri": "https://github.com/spring-cloud-services-samples/cook-config", "label": "master" } }' p-config-server standard config-server

(OR)

using json file:

cf create-service p-config-server standard config-server -c data.json

data.json file contains:

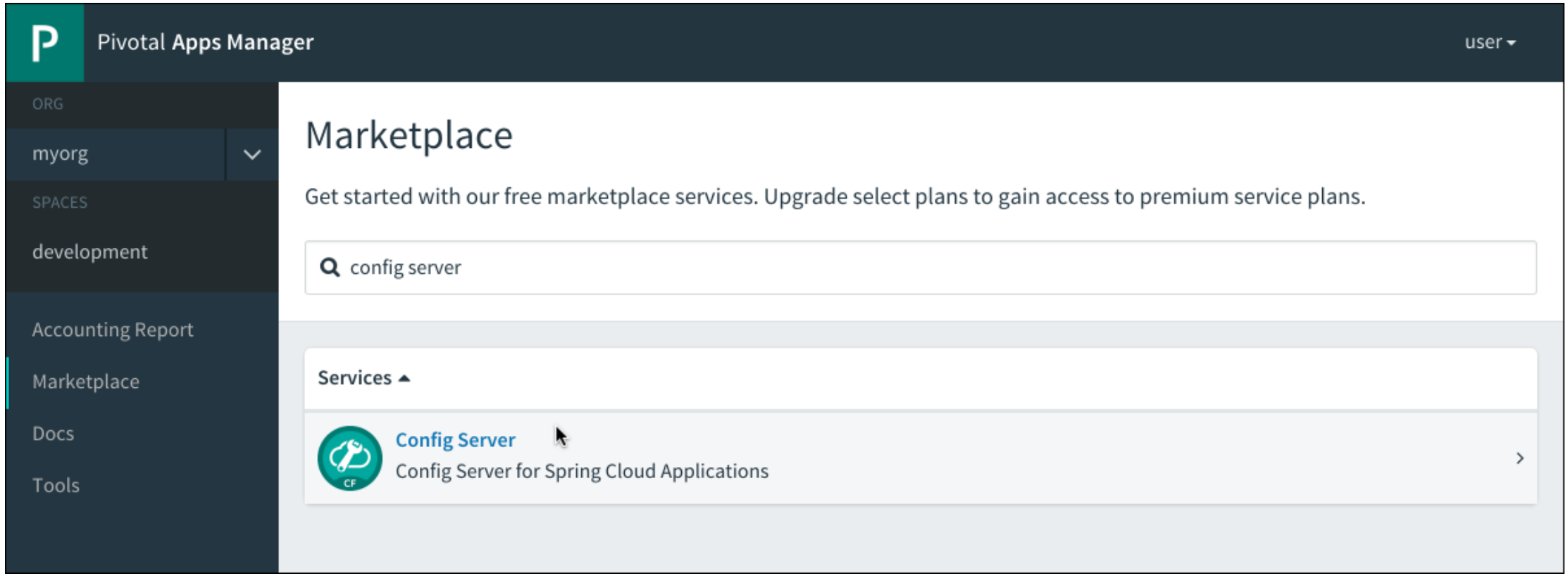
```
{

"git": {
"uri": "https://github.com/spring-cloud-services-samples/cook-config",
"label": "master"
        }
 }
```

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

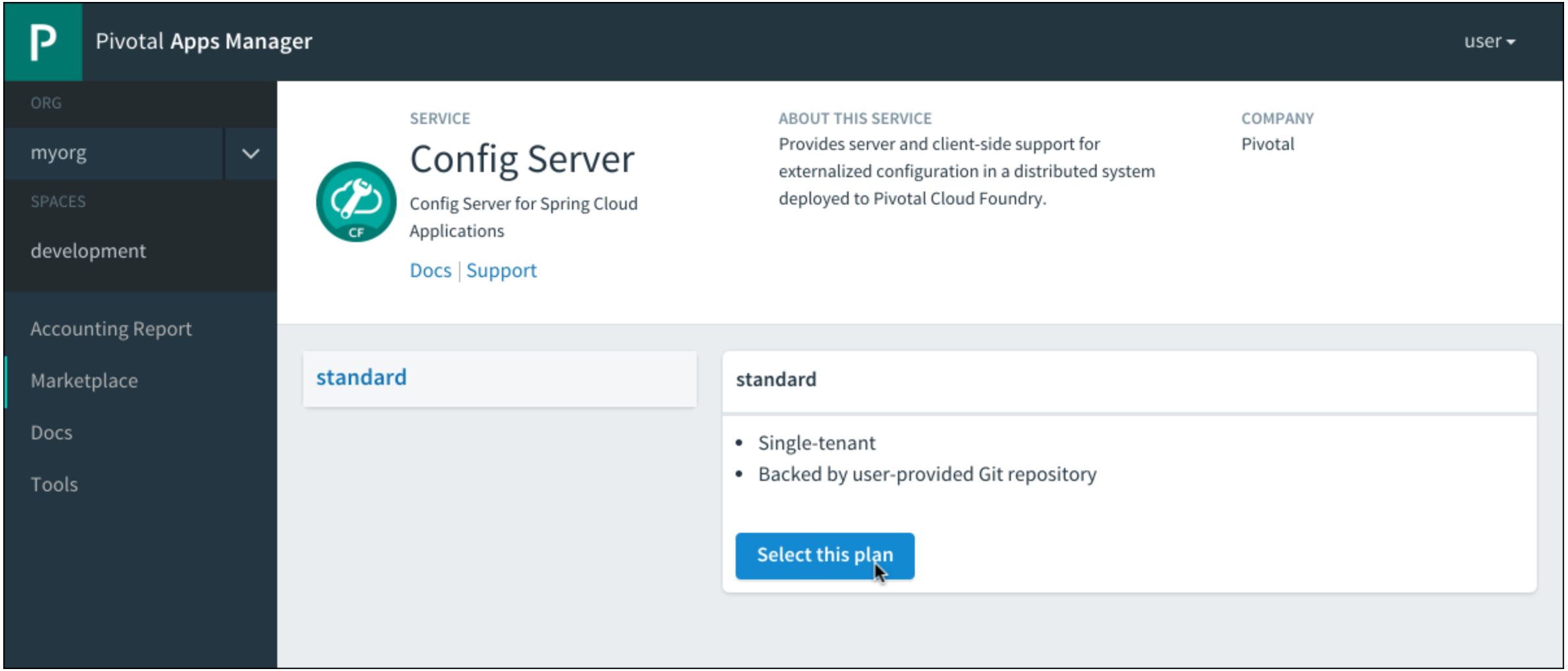
```
$ cf create-service -c '{"git": { "uri": "https://github.com/spring-cloud-samples/config-repo", "repos": { "cook": { "pattern": "cook*", "uri": "https://github.com/spring-cloud-services-samples/cook-config" } } }, "count": 3 }' p-config-server standard config-server
```

Using Apps Manager

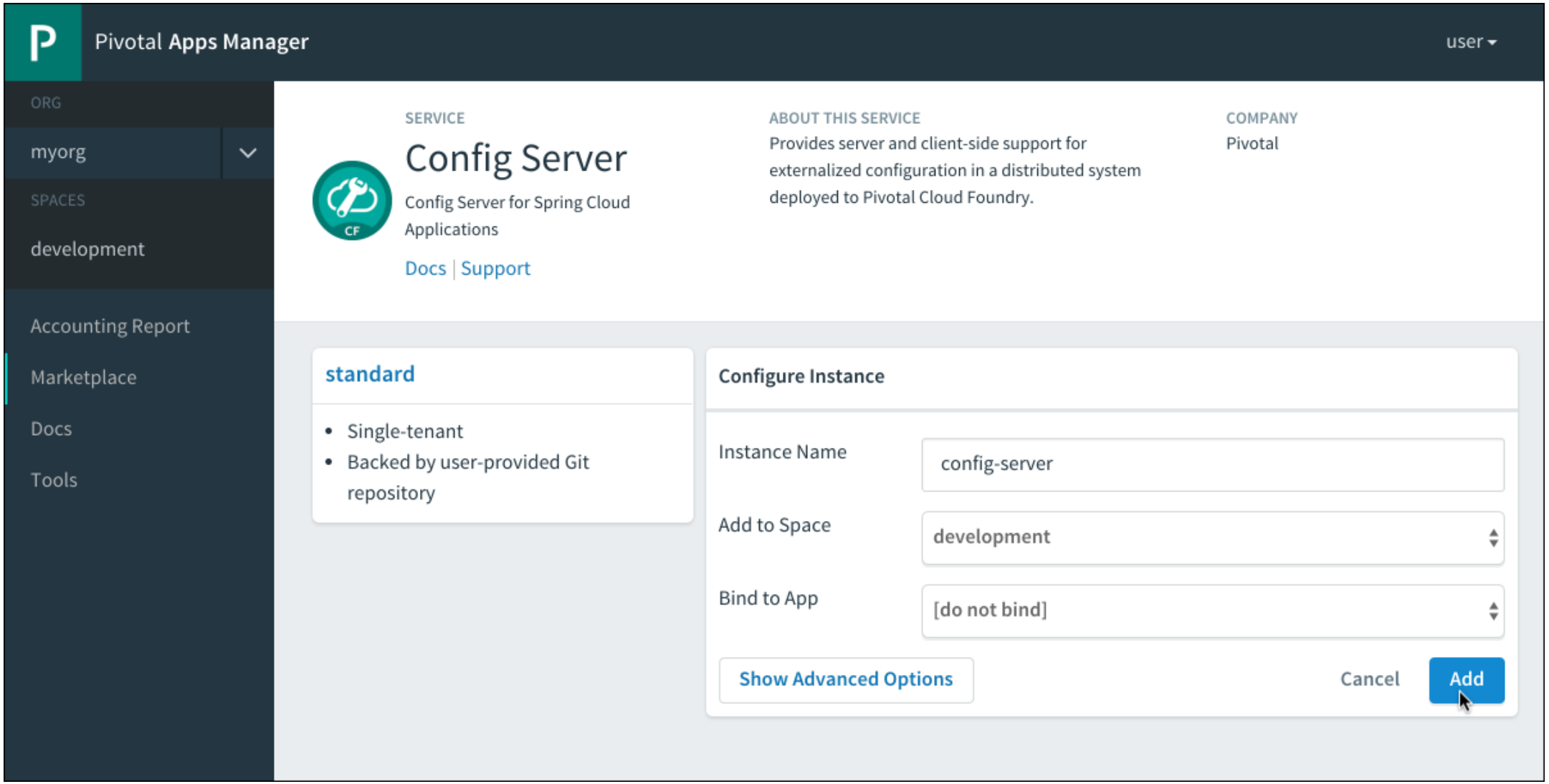
Log into Apps Manager as a Space Developer. In the Marketplace, select Config Server



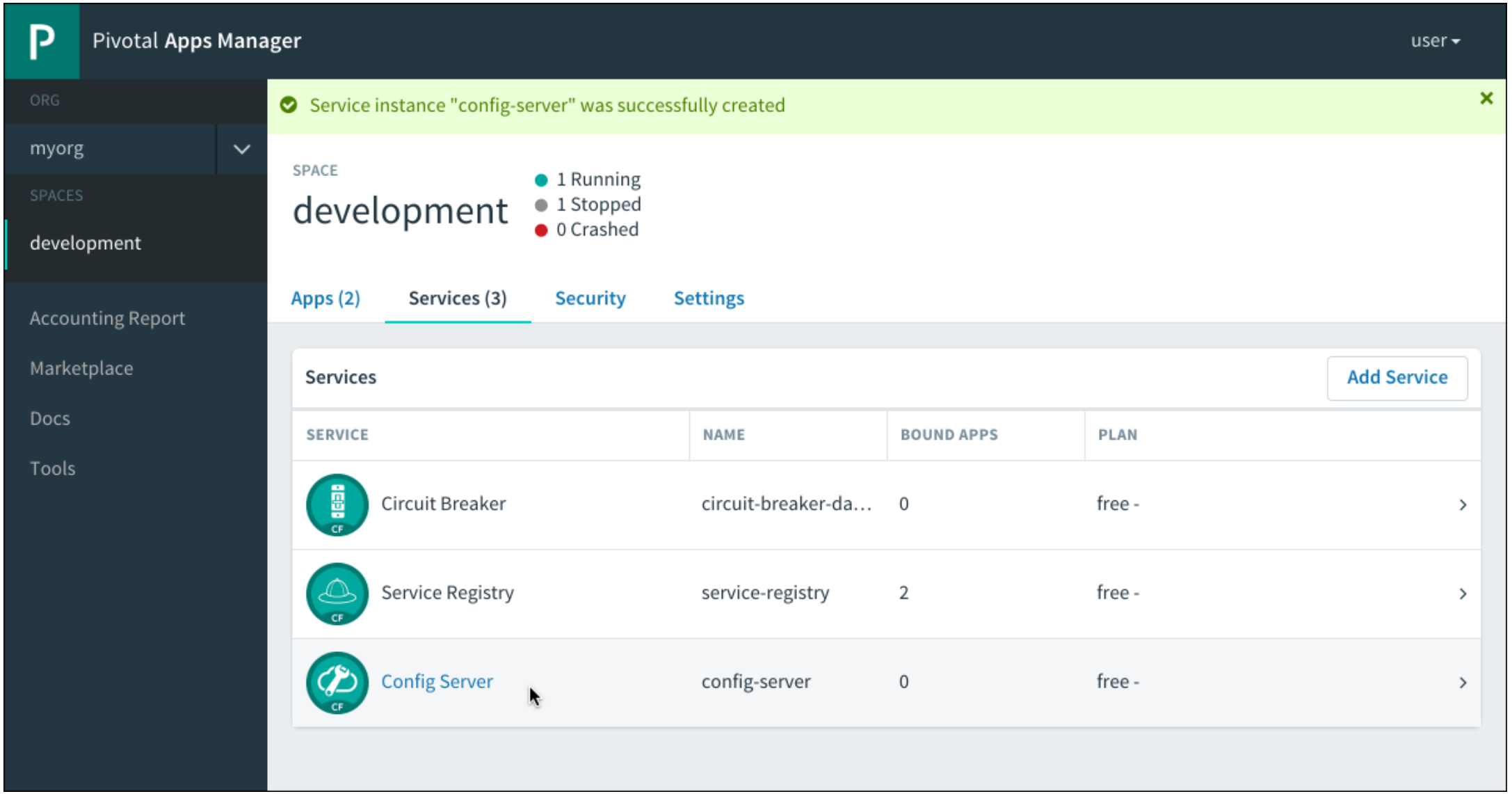
Select the desired plan for the new service instance.



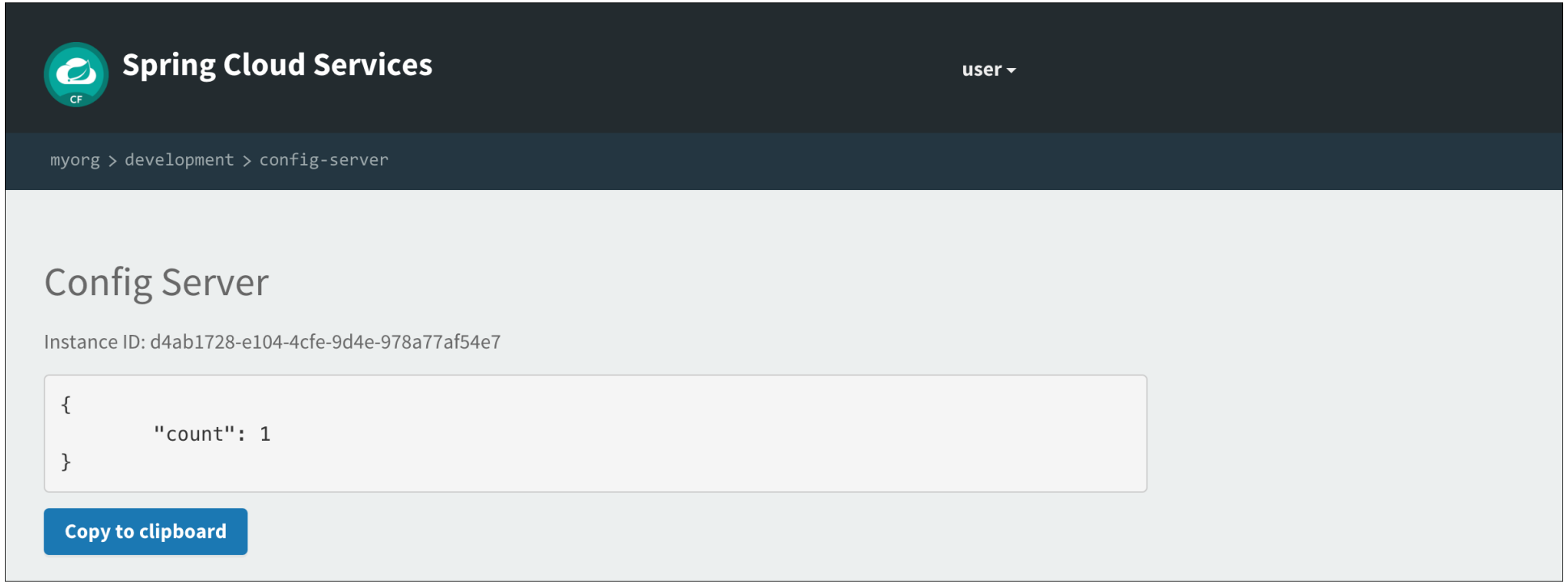
Provide a name for the service instance (for example, “config-server”). Click the Add button.



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



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



Updating an Instance

cf update-service SERVICE_NAME -c '{ "PARAMETER": "VALUE" }’,
where SERVICE_NAME is the name of the service instance
PARAMETER is a supported parameter
VALUE is the value for the parameter

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

Configuring with Git

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

Spring Cloud Config provides a Git backend so that the Spring Cloud Config Server can serve configuration stored in Git.

The Spring Cloud Services Config Server supports this backend and can serve configuration stored in Git to client applications when given the URL to a Git repository (for example, the URL of a repository hosted on GitHub or Bitbucket).

General Configuration

Parameters used to configure configuration sources are part of a JSON object called git, as in {"git": { "uri": "http://example.com/config" } }

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

Configuring with Vault

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

General Configuration

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

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

Declarative Composite Backends

The Spring Cloud Services Config Server provides the ability to serve configuration properties from a composite of multiple backends, such as from multiple GitHub repositories and a HashiCorp Vault server.

This feature builds upon the Composite Environment Repositories.

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

Configuration Properties

The Config Server can serve configuration properties from either Git or HashiCorp Vault configuration sources.

Configuration properties can be applicable to all applications that use the Config Server, specific to an application, or specific to a Spring application profile, and can be stored in encrypted form.

Global Configuration

We can store configuration properties so that they are served to all applications which use the Config Server.

In the configuration repository, a file named application.yml or application.properties contains configuration which will be served to all applications that access the Config Server.

Application-Specific Configuration

We can store configuration properties so that they are served only to a specific application.

In the configuration repository, a file named [APP-NAME].yml or [APP-NAME].properties, where [APP-NAME] is the name of an application, contains configuration which will be served only to the APP-NAME application

Profile-Specific Configuration

We can store configuration properties so that they are served only to applications which have activated a specific Spring application profile.

In the configuration repository, a file named [APP-NAME]-[PROFILE-NAME].yml or [APP-NAME]-[PROFILE-NAME].properties, where [APP-NAME] is the name of an application and [PROFILE-NAME] is the name of an application profile

Encrypted Configuration

We can store configuration properties in encrypted form and have these properties decrypted by the Config Server before they are served to applications.

In a file within the configuration repository, properties whose values are prefixed with {cipher} will be decrypted before they are served to client applications.

To use this feature, you must configure the Config Server with an encryption key.

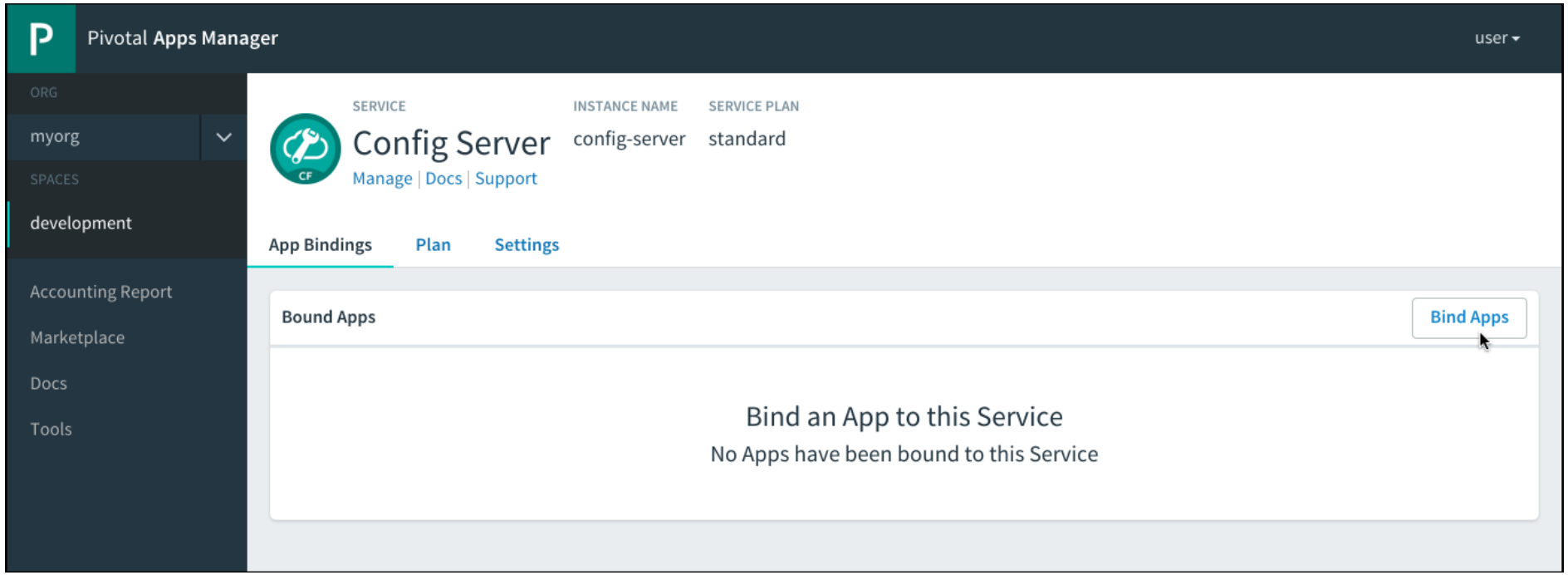
An example of an encrypted property value in an application.yml file:

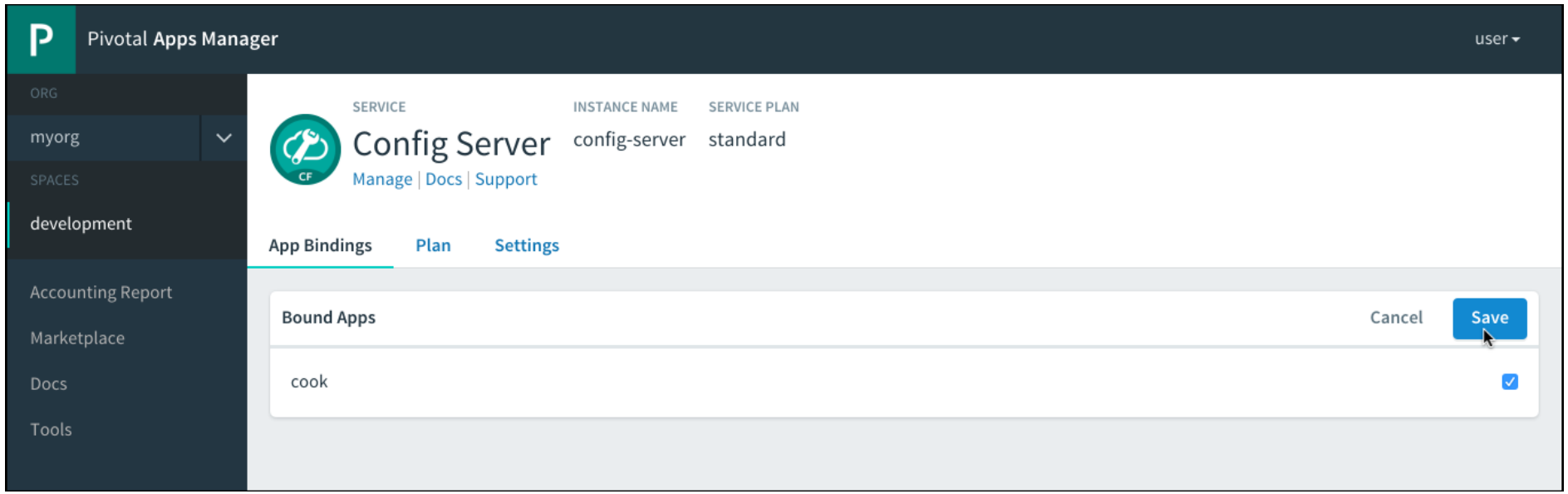
secretMenu:
'{cipher}AQA90Q3GIRAMu6ToMqwS++En2iFzMXIWX99G66yaZFRHrQNq64CntqOzWymd
3xE7uJp
ZKQc9XBIkfyRz/HUGhXRdf3KZQ9bqclwmR5vkiLmN9DHlAxS+6biT+7f8ptKo3fzQ0gGOBaR4
kTnWLBxmVaIkjq1
Qze4aIgsgUWuhbEek+3znkH9+Mc+5zNPvwN8hhgDMDVzgZLB+4YnvWJAq3Au4wEevakAH
HxVY0mXcxj1Ro+H+Zel
IzfF8K2AvC3vmvlmxy9Y49Zjx0RhMzUx17eh3mAB8UMMRJZyUG2a2uGCXmz+UunTA5n/d
WWOvR3VcZyzXPFSFkhN
ekw3db9XZ7goceJSPrRN+5s+GjLCPr+KSnhLmUt1XAScMeqTieNCHT5I='

Configuration Clients

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

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


Pivotal Apps Manager
user
ORG
myorg
SPACES
development
Accounting Report
Marketplace
Docs
Tools
SERVICE
Config Server
Manage | Docs | Support
INSTANCE NAME
config-server
SERVICE PLAN
standard
App Bindings
Plan
Settings
Bound Apps
Bind Apps
Bind an App to this Service
No Apps have been bound to this Service


Pivotal Apps Manager
user
ORG
myorg
SPACES
development
Accounting Report
Marketplace
Docs
Tools
SERVICE
Config Server
Manage | Docs | Support
INSTANCE NAME
config-server
SERVICE PLAN
standard
App Bindings
Plan
Settings
Bound Apps
Cancel
Save
cook

Cloud Foundry Command Line Interface tool (cf CLI)

$ cf bind-service cook config-server

Binding service config-server to app cook in org myorg / space development as admin...
OK

TIP: Use 'cf restage cook' to ensure your env variable changes take effect

Writing Client Applications

pom.xml

Our 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>
```

Add Self-Signed SSL Certificate to JVM Truststore

$ cf set-env cook TRUST_CERTS api.cf.wise.com

Setting env variable 'TRUST_CERTS' to 'api.cf.wise.com' for app cook in org myorg / space development as user...
OK

TIP: Use 'cf restage' to ensure your env variable changes take effect

$ cf restage cook

Use Configuration Values

When the application requests a configuration from the Config Server, it will use a path containing the application name. We 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;
  }

  //...

}
```

Vary Configurations Based on Profiles

We 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.

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

Spring Boot Actuator Dependency

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.

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

We can now visit /env to see the application environment's

```
$ curl http://cookie.apps.wise.com/env
```

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.

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

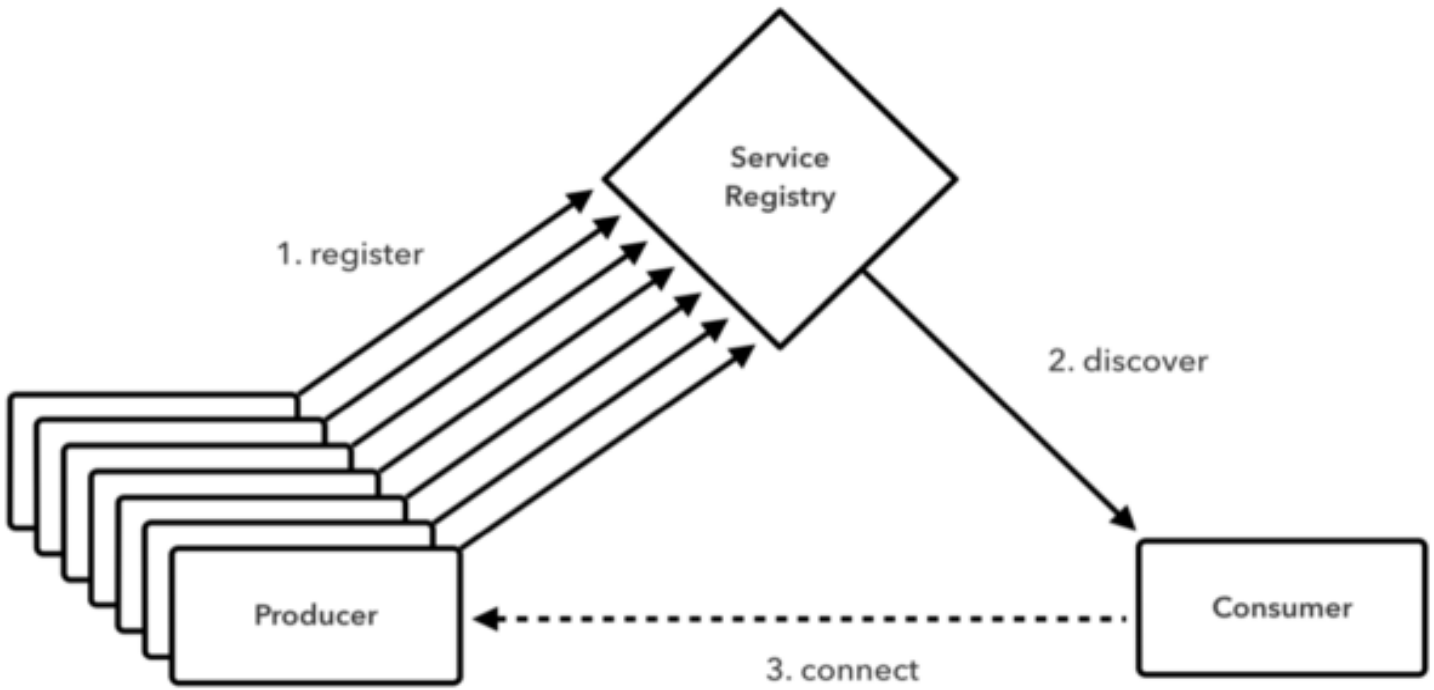
Spring Cloud Connectors

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.

Spring Cloud Connectors provides a simple abstraction for JVM-based applications running on cloud platforms to discover bound services and deployment information at runtime, and provides support for registering discovered services as Spring beans

Service Registry for Pivotal Cloud Foundry

Service Registry for Pivotal Cloud Foundry (PCF) provides the applications with an implementation of the Service Discovery pattern, one of the key tenets of a microservice-based architecture.



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.

Enabling Peer Replication

We can configure a Service Registry service instance to replicate service registrations with a peer Service Registry service instance.

This functionality supports two models:

Peer replication across separate Pivotal Cloud Foundry (PCF) deployments:

We can configure peer replication to allow access to services registered with a Service Registry service instance in a PCF deployment located in a separate datacenter.

Peer replication across PCF organizations and spaces:

We can configure peer replication to allow access to services registered with a Service Registry service instance in another organization or space within the same PCF deployment.

Configuration Parameters

To enable peer replication for a Service Registry service instance, we must specify the peer Service Registry instance's URI using the peers JSON array, which contains an object for each Service Registry peer.

We can find a Service Registry service instance's URI on its dashboard

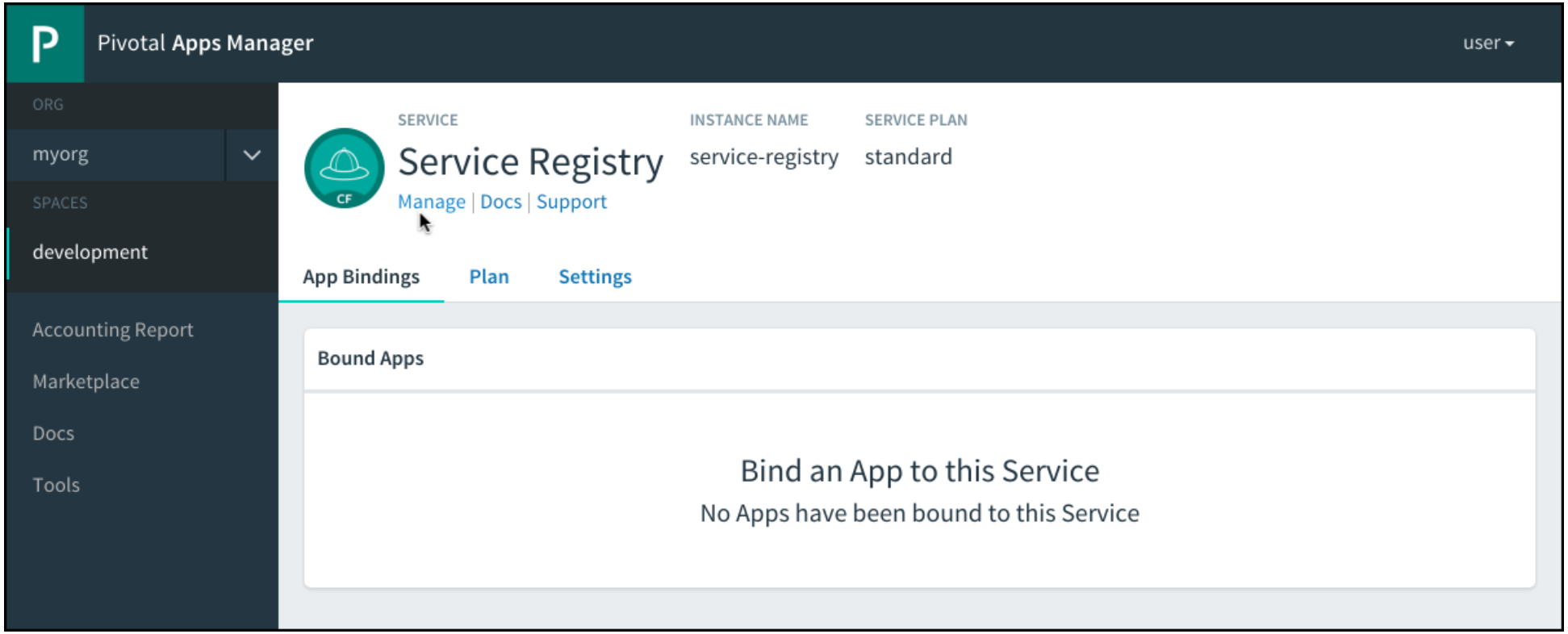
A Service Registry peer can be expressed as shown in the following JSON:

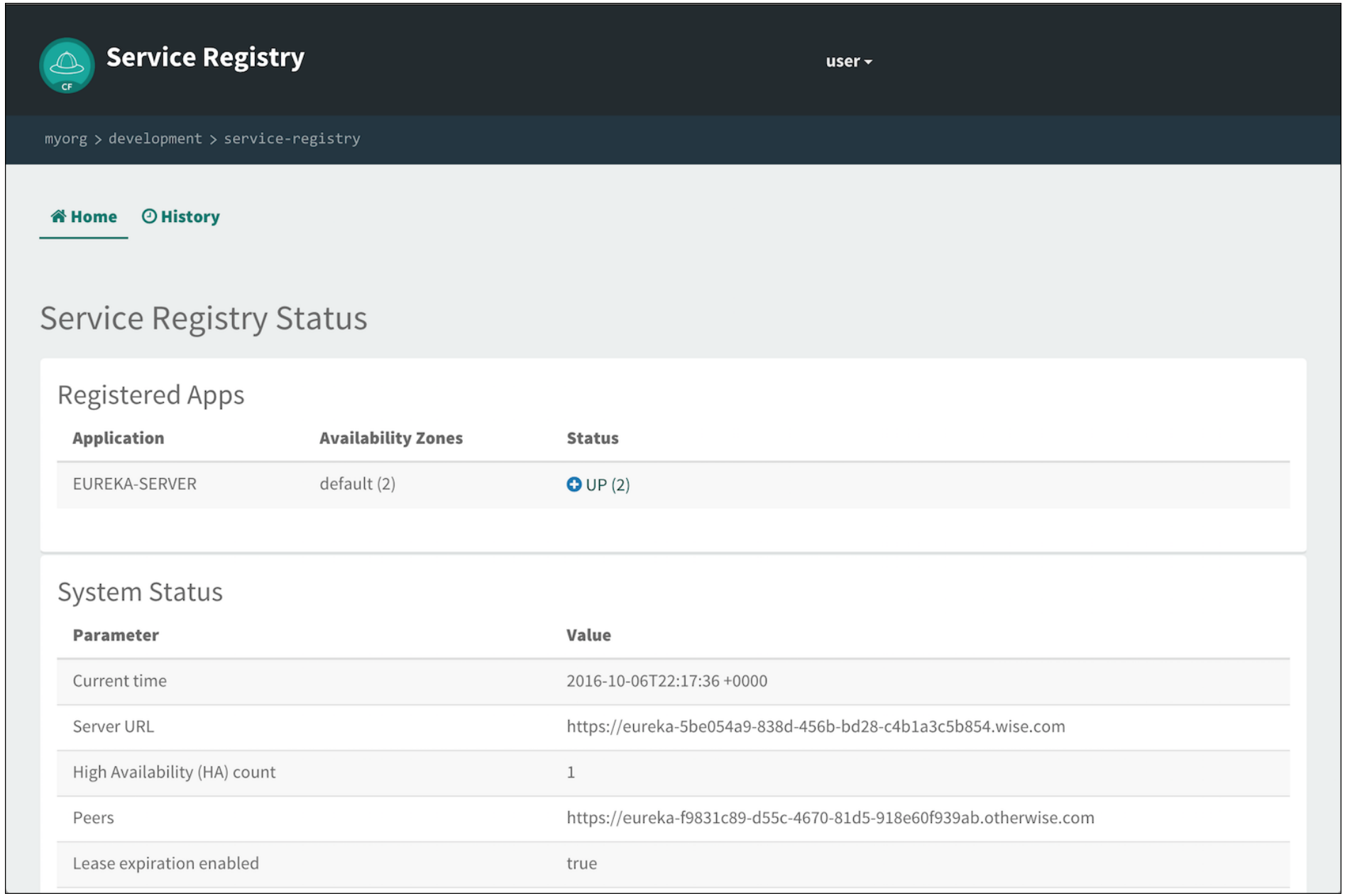
```
{
  "peers": [
    { "uri": "https://eureka-e280160b-d3e3-41ad-93a6-
479f9b298ca6.wise2.com" }
  ]
}
```

```
$ cf create-service p-service-registry standard service-registry -c '{ "peers": [ {"uri": "https://eureka-e280160b-d3e3-41ad-93a6-479f9b298ca6.wise2.com"} ] }’
```

```
$ cf update-service service-registry -c '{ "peers": [ {"uri": "https://eureka-e280160b-d3e3-41ad-93a6-479f9b298ca6.wise2.com"} ] }'
```

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




Service Registry
user
myorg > development > service-registry
Home
History
Service Registry Status
Registered Apps
Application
Availability Zones
Status
EUREKA-SERVER
default (2)
UP (2)
System Status
Parameter
Value
Current time
2016-10-06T22:17:36 +0000
Server URL
https://eureka-5be054a9-838d-456b-bd28-c4b1a3c5b854.wise.com
High Availability (HA) count
1
Peers
https://eureka-f9831c89-d55c-4670-81d5-918e60f939ab.otherwise.com
Lease expiration enabled
true

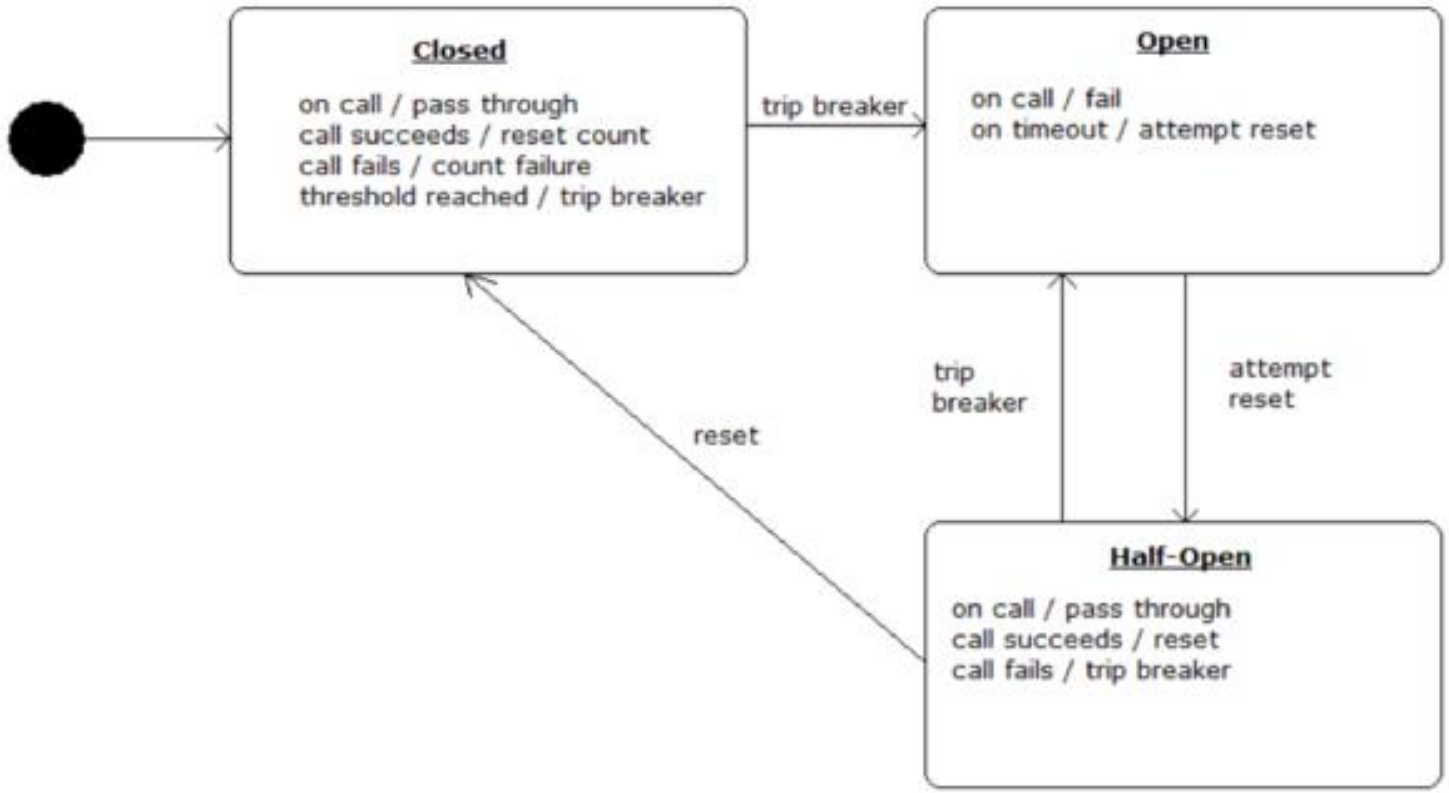
Circuit Breaker Dashboard

Circuit Breaker Dashboard provides Spring applications with an implementation of the Circuit Breaker pattern.

Cloud-native architectures are typically composed of multiple layers of distributed services.

End-user requests may comprise multiple calls to these services, and if a lower-level service fails, the failure can cascade up to the end user and spread to other dependent services.

Heavy traffic to a failing service can also make it difficult to repair. Using Circuit Breaker Dashboard, we can prevent failures from cascading and provide fallback behavior until a failing service is restored to normal operation.


Closed
on call / pass through
call succeeds / reset count
call fails / count failure
threshold reached / trip breaker
trip breaker
Open
on call / fail
on timeout / attempt reset
trip
breaker
attempt
reset
reset
Half-Open
on call / pass through
call succeeds / reset
call fails / trip breaker

When applied to a service, a circuit breaker watches for failing calls to the service. If failures reach a certain threshold, it “opens” the circuit and automatically redirects calls to the specified fallback mechanism.
This gives the failing service time to recover.

Circuit Breaker Dashboard is based on Hystrix, Netflix’s latency and fault-tolerance library

Client Dependencies

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>
```

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);
   }
```

Use a Circuit Breaker with a Feign Client

We cannot apply @HystrixCommand directly to a Feign client interface at this time. Instead, we 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.

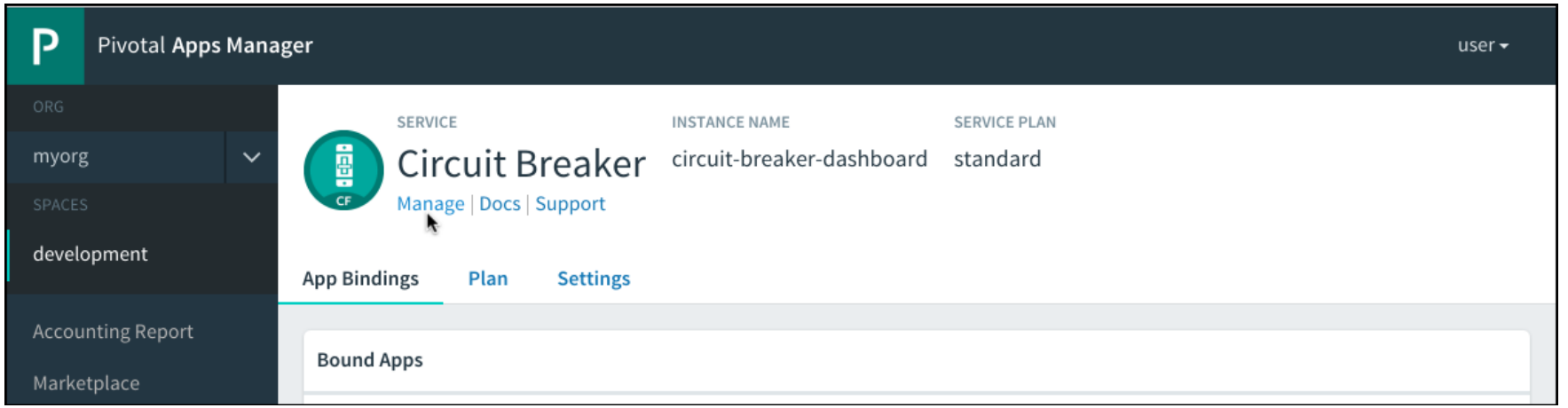
AgencyApplication class is annotated with @EnableFeignClients.

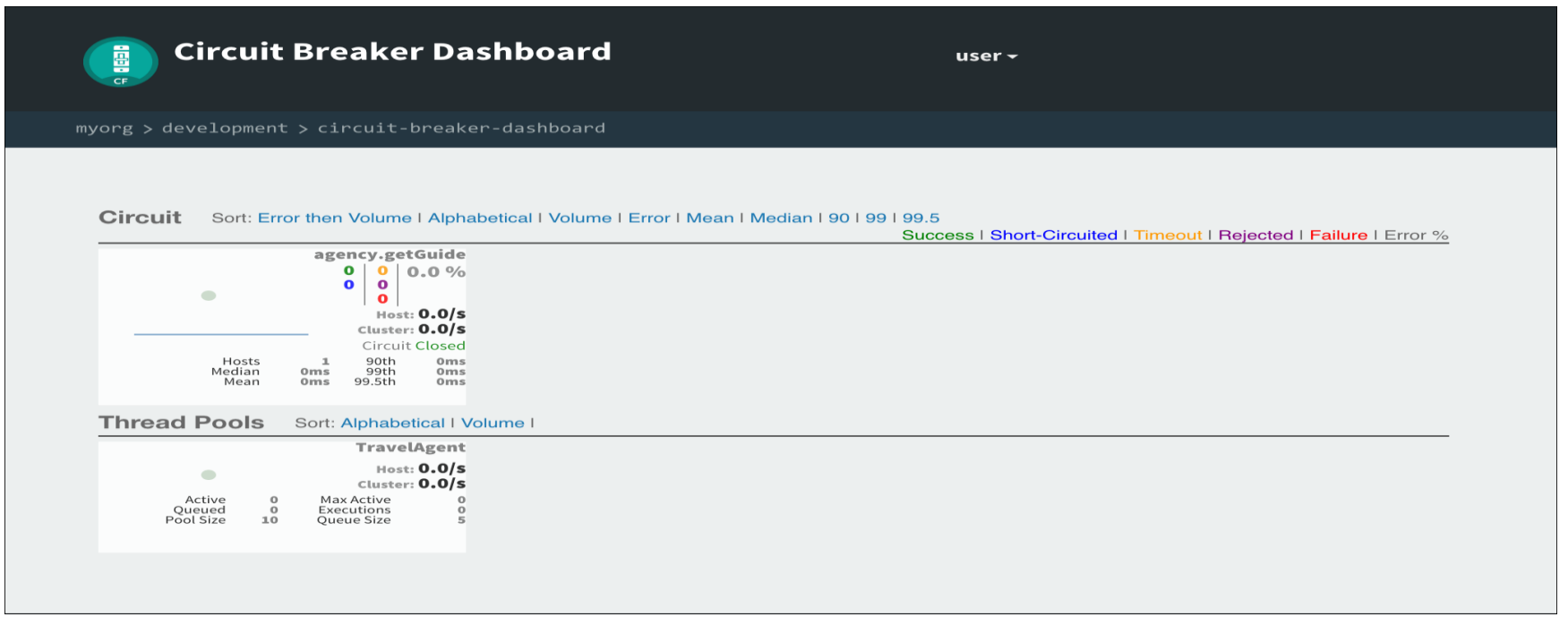
```
import org.springframework.cloud.netflix.feign.EnableFeignClients;

@SpringBootApplication
@EnableDiscoveryClient
@RestController
@EnableCircuitBreaker
@EnableFeignClients
public class AgencyApplication {
```

The application has a Feign client called CompanyClient.

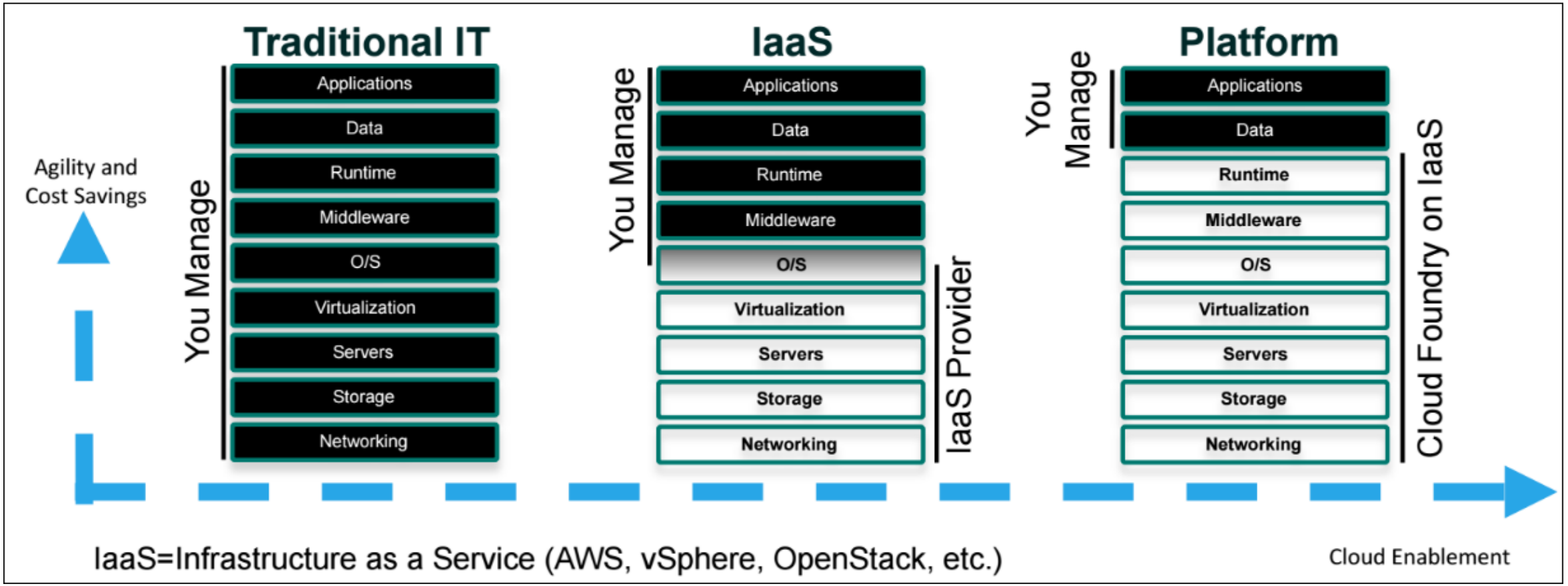
```
package agency;

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();
}
```

Using the Dashboard


Pivotal Apps Manager
user
ORG
myorg
SPACES
development
Accounting Report
Marketplace
SERVICE
Circuit Breaker
Manage | Docs | Support
INSTANCE NAME
circuit-breaker-dashboard
SERVICE PLAN
standard
App Bindings
Plan
Settings
Bound Apps


Circuit Breaker Dashboard
user
myorg > development > circuit-breaker-dashboard
Circuit
Sort: Error then Volume | Alphabetical | Volume | Error | Mean | Median | 90 | 99 | 99.5
Success | Short-Circuited | Timeout | Rejected | Failure | Error %
agency.getGuide
0 0 0.0 %
0 0
0
Host: 0.0/s
Cluster: 0.0/s
Circuit Closed
Hosts 1
Median 0ms
Mean 0ms
90th 0ms
99th 0ms
99.5th 0ms
Thread Pools
Sort: Alphabetical | Volume |
TravelAgent
Host: 0.0/s
Cluster: 0.0/s
Active 0
Queued 0
Pool Size 10
Max Active 0
Executions 0
Queue Size 5

Cloud Foundry Architecture


Traditional IT
Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking
You Manage
IaaS
Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking
You Manage
IaaS Provider
Platform
You Manage
Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking
Cloud Foundry on IaaS
Agility and Cost Savings
IaaS=Infrastructure as a Service (AWS, vSphere, OpenStack, etc.)
Cloud Enablement

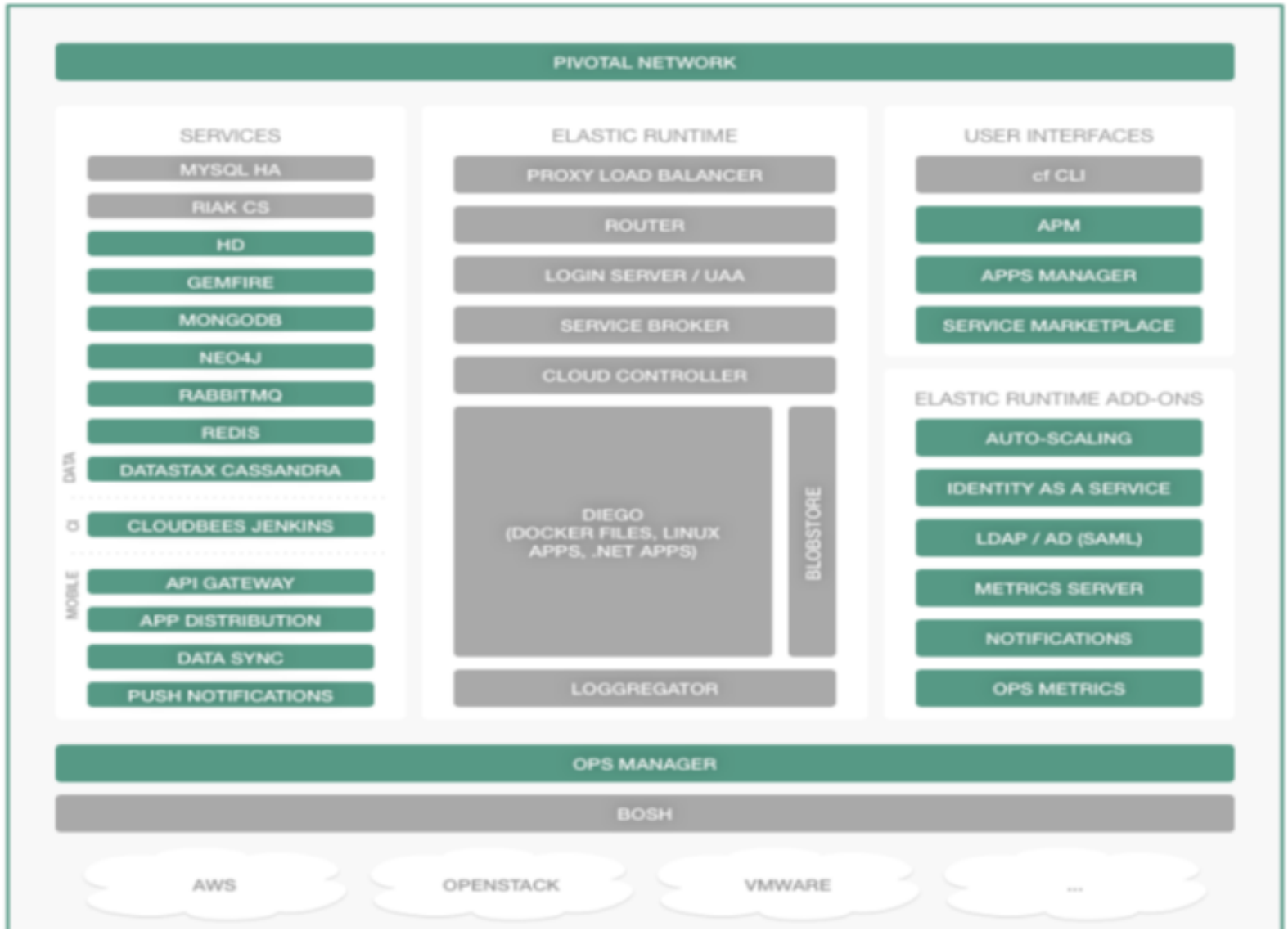
How Cloud Foundry Works

To flexibly serve and scale apps online, Cloud Foundry has subsystems that perform specialized functions.

Pivotal Cloud Foundry Commercialization Model

Open source software provides the basis for the platform

Commercial distribution provides value add features, enterprise services, support, docs, certs, etc.



How the Cloud Balances Its Load

Clouds balance their processing loads over multiple machines, optimizing for efficiency and resilience against point failure.

A Cloud Foundry installation accomplishes this at three levels:

BOSH creates and deploys virtual machines (VMs) on top of a physical computing infrastructure, and deploys and runs Cloud Foundry on top of this cloud. To configure the deployment, BOSH follows a manifest document.

The CF **Cloud Controller** runs the apps and other processes on the cloud's VMs, balancing demand and managing app lifecycles.

The **router** routes incoming traffic from the world to the VMs that are running the apps that the traffic demands, usually working with a customer-provided load balancer.

Component : BOSH

Bosh is a project that unifies release engineering, deployment, and lifecycle management of small and large-scale cloud software.

BOSH can provision and deploy software over hundreds of VMs.

It also performs monitoring, failure recovery, and software updates with zero-to-minimal downtime.

While BOSH was developed to deploy Cloud Foundry PaaS, it can also be used to deploy almost any other software (Hadoop, for instance). BOSH is particularly well-suited for large distributed systems

Component: Cloud Controller

The Cloud Controller provides REST API endpoints for clients to access the system.

The Cloud Controller maintains a database with tables for orgs, spaces, services, user roles, and more.

Diego Auction

The Cloud Controller uses the Diego Auction to balance application processes over the cells in a Cloud Foundry installation.

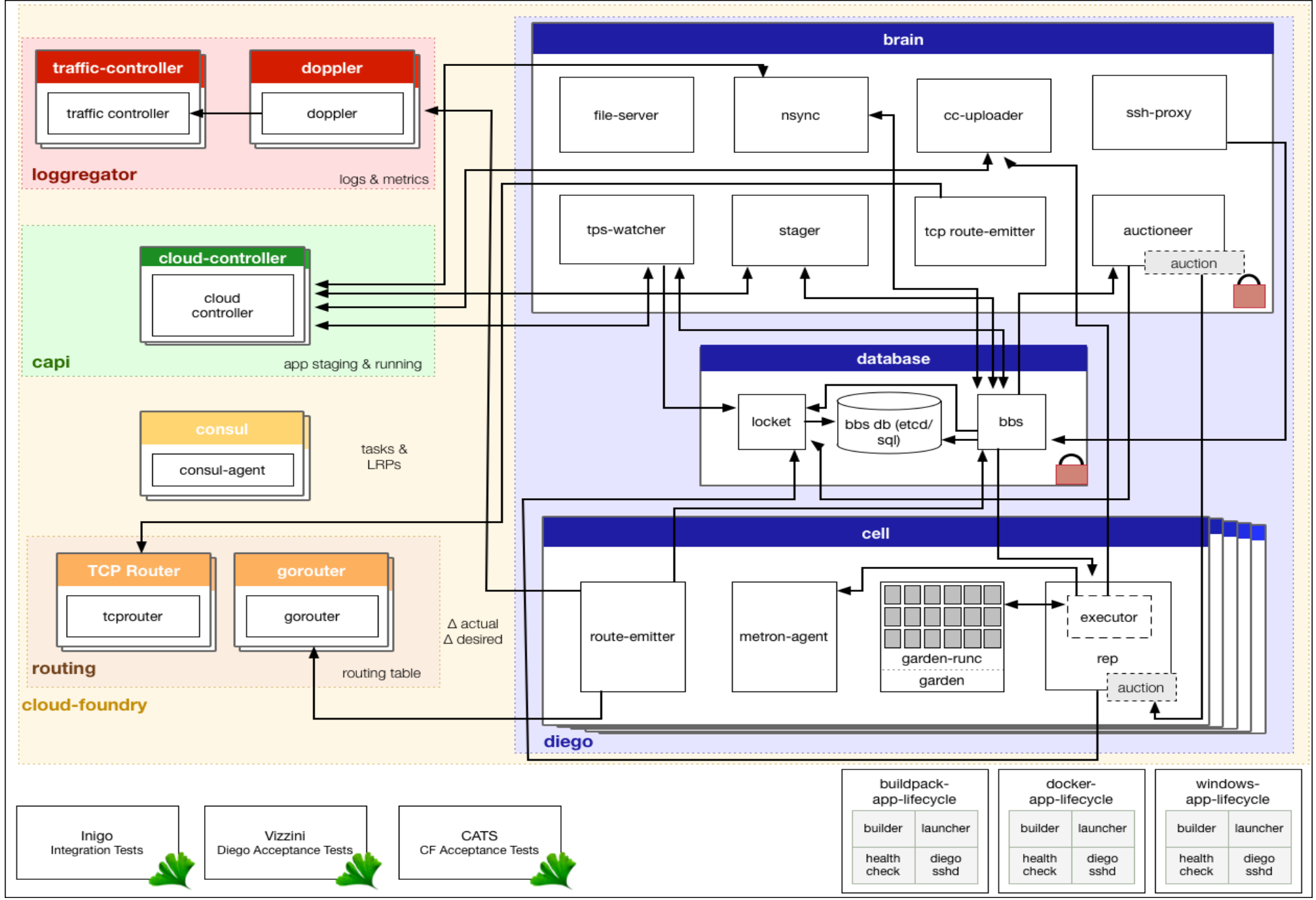
The **Diego** Auction balances application processes, also called jobs, over the virtual machines (VMs) in a Cloud Foundry installation.

When new processes need to be allocated to VMs, the Diego Auction determines which ones should run on which machines.

The auction algorithm balances the load on VMs and optimizes application availability and resilience. This topic explains how the Diego Auction works at a conceptual level.

Component: Gorouter

The Gorouter routes traffic coming into Cloud Foundry to the appropriate component, whether it is an operator addressing the Cloud Controller or an application user accessing an app running on a Diego Cell.


traffic-controller
traffic controller
doppler
doppler
loggregator
logs & metrics
cloud-controller
cloud controller
capi
app staging & running
consul
consul-agent
tasks & LRPs
TCP Router
tcprouter
gorouter
gorouter
routing
routing table
Δ actual
Δ desired
cloud-foundry
brain
file-server
nsync
cc-uploader
ssh-proxy
tps-watcher
stager
tcp route-emitter
auctioneer
auction
database
locket
bbs db (etcd/ sql)
bbs
cell
route-emitter
metron-agent
garden-runc
garden
executor
rep
auction
diego
Inigo
Integration Tests
Vizzini
Diego Acceptance Tests
CATS
CF Acceptance Tests
buildpack-app-lifecycle
builder
launcher
health check
diego sshd
docker-app-lifecycle
builder
launcher
health check
diego sshd
windows-app-lifecycle
builder
launcher
health check
diego sshd

Diego Brain

Diego Brain components distribute Tasks and LRPs (Long Running Process) to Diego Cells, and correct inconsistency between ActualLRP and DesiredLRP counts to ensure fault-tolerance and long-term consistency.

The Diego Brain consists of the Auctioneer.

Auctioneer

Uses the auction package to run Diego Auctions for Tasks and LRPs
Communicates with Cell Reps over HTTP

Maintains a lock in the BBS(Bulletin Board System) that restricts auctions to one Auctioneer at a time

Diego Cell Components

Diego Cell components manage and maintain Tasks and LRPs.

Rep
Represents a Cell in Diego Auctions for Tasks and LRPs
Mediates all communication between the Cell and the BBS

Executor
Runs as a logical process inside the Rep

Garden
Provides a platform-independent server and clients to manage Garden containers

Metron Agent
Forwards application logs, errors, and application and Diego metrics to the Loggregator Doppler component

Loggregator is the logging system used in CloudFoundry.

Loggregator Goals:

Real time streaming of logs
Producers do not experience backpressure
Logs can be routed to several consumers
Elastic and horizontal scalability
High message reliability
Low latency
Flexibile consumption
Security via gRPC with mutual TLS
Opinionated log structure