Architecture

## [Cloud Foundry HA with NATS and other explaination by James Bayer](http://blog.csdn.net/resouer/article/details/8882334)

There has been another post on this previously. When running on vSphere / SAN, this is generally not an issue as we have relied on vSphere HA features for several years and it offers a robust IaaS that restarts any SPOF VMs immediately. This is how we have been running CloudFoundry.com for about 2 years without meaningful downtime related to single points of failure. If you do not have HA IaaS with resilient storage, then Cloud Foundry is not fully multi-site or HA with out of the box configurations yet. We are working on removing all single points of failure for environments like AWS that do not have the same capabilities as vSphere.

- We have recently worked on MySQL support for Cloud Controller DB, which means that when running on AWS that RDS could be used.

- There has also been some discussions about removing single points of failure in NATS recently on the GitHub issues: <https://github.com/cloudfoundry/cf-release/issues/32>

- Health Manager is currently a SPOF

- UAA 1.4.x (almost deployed) will support horizontal scaling

So we are actively working on this, but we do not have all the pieces finished. We will be updating the [cloudfoundry.github.com](http://cloudfoundry.github.com/) docs as we get closer.

Health Manager (HM) only operates as a single node and is therefore not HA, but a CF system should operate in degraded mode. In this degraded mode the actual state of the world with respect to application state, instances, etc and the intended state of the world will drift until HM becomes available again.

NATS is still a SPOF but we have completed a bunch of work to make sure CF components behave well when it is not around. Basically, the system should operate in a degraded mode until NATS returns. Previously many components did not behave well when NATS went away. In a subsequent set of work, we will consider other HA options for NATS including things like running a Hot/Warm NATS pair and use something like VRRP to migrate the IP, clustered NATS, or other options which keep NATS available. We decided that planning to lose NATS completely was the better path now for overall system health than try to prevent something that could conceivably happen.

Examples of how this current set of NATS work affects particular CF components:

- All - should attempt to periodically reconnect to NATS instead of exiting or giving up.

- Cloud Controller - In the degraded mode when NATS is unavailable Cloud Controller API requests to make writable changes to apps don't take effect such as push (will fail to stage), scale (should take effect in CC DB but not be communicated to DEAs), delete (not sure what happens here until I try it, but I'd expect to remove the app from CC DB and have HM garbage collect the app later). Some read operations that need NATS like stats will also not work while NATS is unavailable.

- Router - when the router cannot reach NATS it will not expire the routes it knows about so existing apps will continue getting routed to.

- Health Manager - when the HM cannot connect to NATS, start/kill commands should not be evaluated until the NATS connection can be restored.

- DEAs - when the DEA cannot connect to NATS apps should not be stopped and the DEA should not exit.

# Cloud Controller

The Cloud Controller is written in Ruby and provides REST API endpoints for clients to access the system. The Cloud Controller maintains a database with tables for orgs, spaces, apps, services, service instances, user roles, and more.

### Database (CC\_DB)

The Cloud Controller database has been tested with Postgres.

### Blob Store

The Cloud Controller manages a blob store for:

* resources - files that are uploaded to the Cloud Controller with a unique SHA such that they can be reused without re-uploading the file
* app packages - unstaged files that represent an application
* droplets - the result of taking an app package and staging it (processesing a buildpack) and getting it ready to run

The blob store uses [FOG][<http://fog.io/>] such that it can use abstractions like Amazon S3 or an NFS-mounted file system for storage.

## NATS Messaging

The Cloud Controller interacts with other core components of the Cloud Foundry platform using the NATS message bus. For example, it performs the following using NATS:

* Instructs a DEA to stage an application (processes a buildpack for the app) to prepare it to run
* Instructs a DEA to start or stop an application
* Receives information from the Health Manager about applications
* Subscribes to Service Gateways that advertise available services
* Instructs Service Gateways to handle provisioning, unprovision, bind and unbind operations for services

## Testing

By default rspec will run test suite with sqlite3 in-memory database; however, you can specify connection string via DB\_CONNECTION environment variable to test against postgres and mysql. Examples:

DB\_CONNECTION="postgres:**//**postgres@localhost:5432**/**ccng" rspec

DB\_CONNECTION="mysql2:**//**root:password@localhost:3306**/**ccng" rspec

Travis currently runs 3 build jobs against sqlite, postgres, and mysql.

## Logs

Cloud Controller uses [Steno](http://github.com/cloudfoundry/steno) to manage its logs. Each log entry includes a “source” field to designate which module in the code the entry originates from. Some of the possible sources are 'cc.app', 'cc.app\_stager', 'cc.dea.client' and 'cc.healthmanager.client'.

## Configuration

The Cloud Controller uses a YAML configuration file. For an example, see config/cloud\_controller.yml. Some of the keys that are read from this configuration file are:

* logging - a [steno configuration hash](http://github.com/cloudfoundry/steno#from-yaml-file)
* bulk\_api - basic auth credentials for the application state bulk API. In Cloud Foundry, this endpoint is used by the health manager to retrieve the expected state of every user application.
* uaa - URL and credentials for connecting to the [UAA](http://github.com/cloudfoundry/uaa), Cloud Foundry's OAuth 2.0 server

# Execution Agent (DEA)

This page should describe the DEA

At the bottom of the description should be a short reference to NATS messages per <http://apidocs.cloudfoundry.com/>

# Health Manager

Health Manager monitors the state of the applications and ensures that started applications are indeed running, their versions and number of instances correct.

Conceptually, this is done by maintaining a Known State of applications and comparing it against the Expected State. When discrepancies are found, actions are initiated to bring the applications to the Expected State, e.g., start/stop commands are issued for missing/extra instances, respectively.

Additionally, Health Manager collects and exposes statistics and health status for individual applications, as well as aggregates for frameworks, runtimes, etc.

## AppState

The state of each application is represented by an instance of an aptly named class AppState. AppState gets forwarded important state-changing messages (i.e. hearbeats and exit signals), updates its internal state accordingly and then invokes registered event handlers. It is the job of these handlers (housed in the Harmonizer, see below) to enforce complex policies, e.g., whether to restart application, if so, with which priority, etc.

## Components

HM is comprised of the following components:

* Manager
* Harmonizer
* Scheduler
* ExpectedStateProvider
* KnownStateProvider
* Nudger
* Reporter
* Shadower

### Manager

Provides an entry point, configures, initializes and registers other components.

### Harmonizer

Expresses the policy of bringing the applications to the Expected State by observing the Known State.

Harmonizer sets up the interactions between other components, and aims to achieve clarity of the intent through delegation:

Known State and Expected State are compared periodically with the use of the Scheduler and Nudger actions are Scheduled to bring the States into harmony.

### Scheduler

Encapsulates EventMachine-related functionality such as timer setup and cancellation, quantization of long-running tasks to prevent EM Reactor loop blocking.

### Expected State Provider

Provides the expected state of the application, e.g., whether the application was Started or Stopped, how many instances should be running, etc. This information comes from the Cloud Controller by way of http-based Bulk API, hence the concrete class is BulkBasedExpectedStateProvider.

The Bulk API contains the state of the world as the Cloud Controller says it should be. This is a dump of the CCs database. It might differ from what the world actually looks like, and the Harmonizer will attempt to make the current state match this expected state.

### Known State Provider

The Known state will be discovered from NatsBasedKnownStateProvider, that will listen to heartbeat and other messages on the NATS bus.

The State of each application is represented by an instant of object AppState. That object receives updates of the application state, stores them and notifies registered listeners about events, such as instances\_missing, etc.

### Nudger

Nudger is the interface for health manager to effect the change on the world, by dispatching cloudcontrollers.hm.requests messages that instruct CCs to start or stop instances. Nudger maintains a priority queue of these requests, and deques the messages by a batchful.

### Reporter

Reporter responds to healthmanager.status and healthmanager.health requests.

### Shadower

The Shadower component exists primarily to smoothen the transition from the original HealthManager to this new HealthManager 2.0

The Shadower implements shadow-mode for HM-2, where it quietly and passively observes the state of the world and comes up with harmonization decisions, but rather than publishing harmonization messages on the NATS bus, it keeps track of them. It also listens for harmonization messages (most likely coming from the original HealthManager) and compares those messages with the ones HM-2 has produced. Ideally, these two sets of messages should perfectly overlap. When they don't, shadower issues warnings.

When in shadow-mode, all outgoing messages are sent to the Shadower, rather than being published to NATS.

## Harmonization Policy in Detail

Conceptually, harmonization happens in two ways:

* by reacting to messages (such as droplet.exited);
* by periodically scanning the world and enumerating applications, looking for anomalies.

### droplet.exited signal

There are three distinct scenarios possible when droplet.exited signal arrives:

* application is stopped; means the application was stopped explicitly, no action required;
* DEA evacuation; the DEA is being evacuated and all instances from that DEA need to be restarted somewhere else. HM-2 initiates that restarting;
* application instance crashed; That instance needs to be restarted unless it crashed multiple times in short period of time, in which case it is declared flapping. See more on this below.

### flapping instances

An instance of application is declared flapping if it crashed more than flapping\_death times within flapping\_timeout seconds. There are several possible reasons for flapping:

* app is completely broken and simply does not start;
* app has a bug that results in a crash every once in a while;
* app has a dependency on the external world or a CF-provisioned service, and that dependency is unavailable, perhaps temporarily, resulting in app repeatedly crashing.

Handling flapping apps is hard. We'd like to:

* make the best effort to restart an app, when it makes sense;
* provide the crashlogs for crashing instances;
* cut down on the overhead associated with restarting an app, particularly relating to moving application bits to DEA and storing it there.
* avoid IO spikes due to massive simultaneous restarts

In order to accomodate these conflicting requirements, the following policy for flapping instances (FI) adopted:

* initially the FI is restarted with a delay defined by min\_restart\_delay config value;
* for each subsequent crash, the delay is doubled, but not to exceed max\_restart\_delay config value;
* a random noise is added to the value of delay, its maximum absolute value defined by delay\_time\_noise config value;
* if the number of crashes for a given FI exceeds giveup\_crash\_number, give up restarting attempts. This behavior can be turned off.

### Heartbeat processing

DEAs peridically send out heartbeat messages on NATS bus. These heartbeats contain DEA identifying information, as well as information on application instances running on respective DEAs.

The heartbeats are used to establish “missing” and “extra” indices. Missing indices are then commanded to start, extra indices are commanded to stop.

AppState object tracks heartbeats for each instance of each version.

An instance is “missing” if a live version of this instance has not received a heartbeat in the last droplet\_lost seconds.

However, an instance\_missing event is only triggered if the AppState was not reset recently, and if check\_for\_missing\_instances method has been invoked.

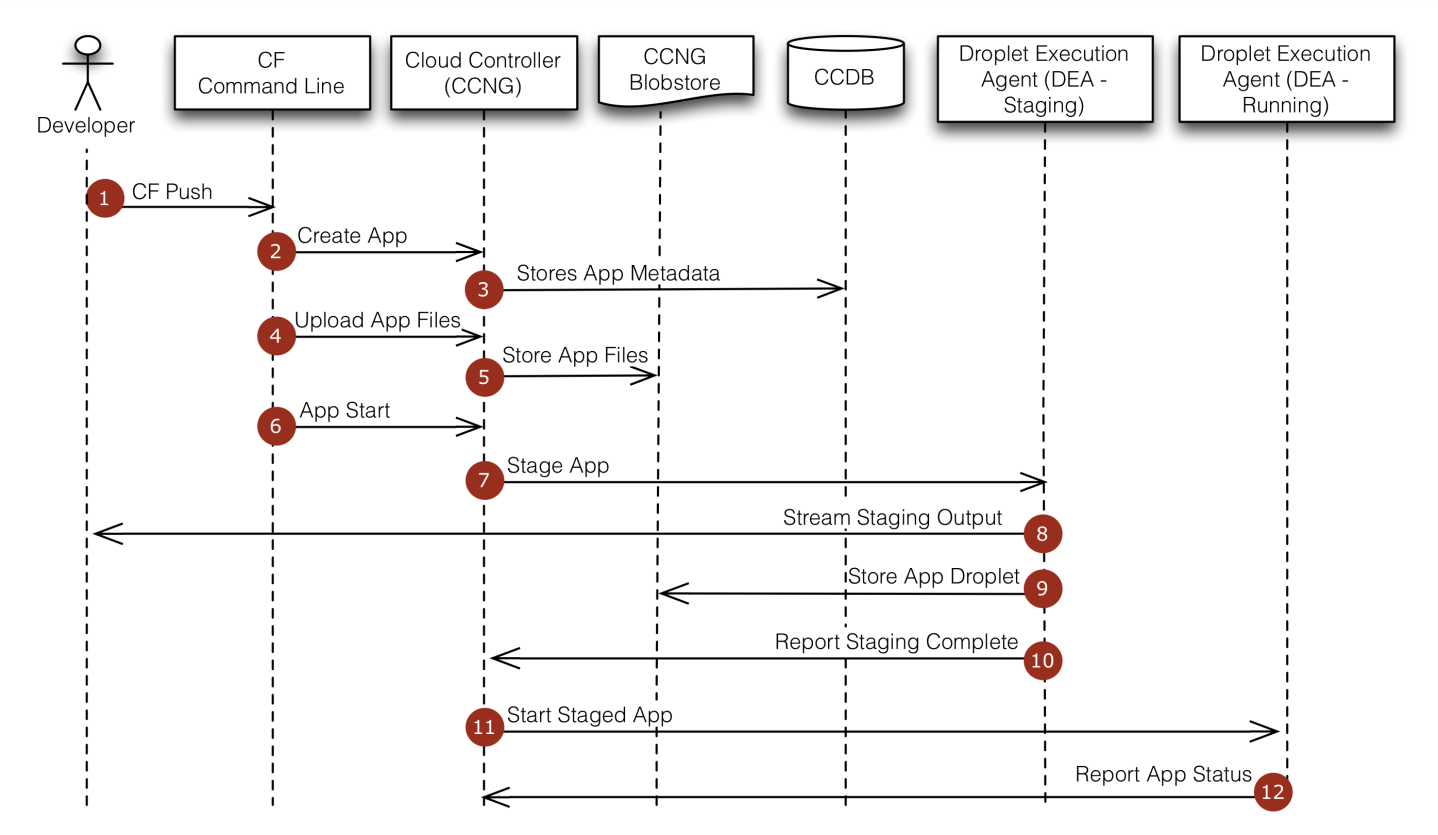
## Configuration

HealthManager reads its configuration from a YAML file. Look at the [example config file](https://github.com/cloudfoundry/health_manager/blob/master/config/health_manager.yml) for an explanation of all the configurable variables.

## Logs

HealthManager uses [Steno](http://github.com/cloudfoundry/steno) to manage its logs. The logging key in the config file provides information for Steno configuration.

# How Applications Are Staged



1. At the command line, the user goes into the directory holding their application and uses the cf command line tool to issue a push command.
2. The cf command line tool tells CCNG to create a record for the application.
3. CCNG stores the application metadata (e.g. the app name, number of instances the user specified, the buildpack and stack).
4. The cf command line tool uploads the application files
5. CCNG stores the raw application files in the blobstore.
6. The cf command line tool issues an app start command.
7. Because the app has not already been staged, CCNG chooses a DEA instance from the DEA pool to stage the application. The staging DEA uses the instructions in the buildpack to stage the application.
8. The staging DEA streams the output of the staging process so the developer can troubleshoot application staging problems.
9. The staging DEA packages up the resulting staged application into a tar ball called a “droplet” and stores it in the blobstore. The results are cached and used next time the application is staged.
10. The staging DEA reports to CCNG that staging is complete.
11. CCNG chooses one or more DEAs from the pool to run the staged application.
12. The running DEAs report back the status of the application to CCNG.

# Messaging (NATS)

This information was adapted from the [Nats](https://github.com/derekcollison/nats) README. Nats is a lightweight publish-subscribe and distributed queueing messaging system written in Ruby.

## Getting Started

**[**sudo**]** gem install nats

**==** or **==**

**[**sudo**]** rake geminstall

nats-sub foo &

nats-pub foo 'Hello World!'

## Basic Usage

require "nats/client"

NATS**.**start **do**

*# Simple Subscriber*

NATS**.**subscribe('foo') { **|**msg**|** puts "Msg received : '#{msg}'" }

*# Simple Publisher*

NATS**.**publish('foo.bar.baz', 'Hello World!')

*# Unsubscribing*

sid **=** NATS**.**subscribe('bar') { **|**msg**|** puts "Msg received : '#{msg}'" }

NATS**.**unsubscribe(sid)

*# Requests*

NATS**.**request('help') { **|**response**|** puts "Got a response: '#{response}'" }

*# Replies*

NATS**.**subscribe('help') { **|**msg, reply**|** NATS**.**publish(reply, "I'll help!") }

*# Stop using NATS.stop, exits EM loop if NATS.start started the loop*

NATS**.**stop

**end**

## Wildcard Subscriptions

*# "\*" matches any token, at any level of the subject.*

NATS**.**subscribe('foo.\*.baz') { **|**msg, reply, sub**|** puts "Msg received on [#{sub}] : '#{msg}'" }

NATS**.**subscribe('foo.bar.\*') { **|**msg, reply, sub**|** puts "Msg received on [#{sub}] : '#{msg}'" }

NATS**.**subscribe('\*.bar.\*') { **|**msg, reply, sub**|** puts "Msg received on [#{sub}] : '#{msg}'" }

*# ">" matches any length of the tail of a subject and can only be the last token*

*# E.g. 'foo.>' will match 'foo.bar', 'foo.bar.baz', 'foo.foo.bar.bax.22'*

NATS**.**subscribe('foo.>') { **|**msg, reply, sub**|** puts "Msg received on [#{sub}] : '#{msg}'" }

## Queues Groups

*# All subscriptions with the same queue name will form a queue group*

*# Each message will be delivered to only one subscriber per queue group, queuing semantics*

*# You can have as many queue groups as you wish*

*# Normal subscribers will continue to work as expected.*

NATS**.**subscribe(subject, :queue **=>** 'job.workers') { **|**msg**|** puts "Received '#{msg}'" }

## Advanced Usage

*# Publish with closure, callback fires when server has processed the message*

NATS**.**publish('foo', 'You done?') { puts 'msg processed!' }

*# Timeouts for subscriptions*

sid **=** NATS**.**subscribe('foo') { received **+=** 1 }

NATS**.**timeout(sid, TIMEOUT\_IN\_SECS) { timeout\_recvd **=** **true** }

*# Timeout unless a certain number of messages have been received*

NATS**.**timeout(sid, TIMEOUT\_IN\_SECS, :expected **=>** 2) { timeout\_recvd **=** **true** }

*# Auto-unsubscribe after MAX\_WANTED messages received*

NATS**.**unsubscribe(sid, MAX\_WANTED)

*# Multiple connections*

NATS**.**subscribe('test') **do** **|**msg**|**

puts "received msg"

NATS**.**stop

**end**

*# Form second connection to send message on*

NATS**.**connect { NATS**.**publish('test', 'Hello World!') }

# (Go)Router

The original router can be found [here](http://github.com/cloudfoundry/router). The original router is backed by nginx, that uses Lua code to connect to a Ruby server that – based on the headers of a client's request – will tell nginx whick backend it should use. The main limitations in this architecture are that nginx does not support non-HTTP (e.g. traffic to services) and non-request/response type traffic (e.g. to support WebSockets), and that it requires a round trip to a Ruby server for every request.

The Go implementation of the Cloud Foundry router is an attempt in solving these limitations. First, with full control over every connection to the router, it can more easily support WebSockets, and other types of traffic (e.g. via HTTP CONNECT). Second, all logic is contained in a single process, removing unnecessary latency.

## Getting started

The following instructions may help you get started with gorouter in a standalone environment.

### Setup

git clone https:**//**github.com**/**cloudfoundry**/**gorouter.git

cd gorouter

git submodule update **--**init

**./**bin**/**go install router**/**router

gem install nats

### Start

*# Start NATS server in daemon mode*

nats**-**server **-**d

*# Start gorouter*

**./**bin**/**router

### Usage

When gorouter is used in Cloud Foundry, it receives route updates via NATS. Routes that haven't been updated in 2 minutes (by default) are pruned. Therefore, to maintain an active route, it needs to be updated at least every 2 minutes. The format of these route updates are as follows:

{

"host": "127.0.0.1",

"port": 4567,

"uris": [

"my\_first\_url.vcap.me",

"my\_second\_url.vcap.me"

],

"tags": {

"another\_key": "another\_value",

"some\_key": "some\_value"

}

}

Such a message can be sent to both the router.register subject to register URIs, and to the router.unregister subject to unregister URIs, respectively.

$ nohup ruby **-**rsinatra **-**e 'get("/") { "Hello!" }' **&**

$ nats**-**pub 'router.register' '{"host":"127.0.0.1","port":4567,"uris":["my\_first\_url.vcap.me","my\_second\_url.vcap.me"],"tags":{"another\_key":"another\_value","some\_key":"some\_value"}}'

Published [router.register] : '{"host":"127.0.0.1","port":4567,"uris":["my\_first\_url.vcap.me","my\_second\_url.vcap.me"],"tags":{"another\_key":"another\_value","some\_key":"some\_value"}}'

$ curl my\_first\_url.vcap.me:8080

Hello!

### Instrumentation

Gorouter provides /varz and /healthz http endpoints for monitoring.

The /routes endpoint returns the entire routing table as JSON. Each route has an associated array of host:port entries.

All of the endpoints require http basic authentication, credentials for which can be acquired through NATS. The port, user and password (pass is the config attribute) can be explicitly set in the gorouter.yml config file's status section.

status**:**

port**:** 8080

user**:** some\_user

pass**:** some\_password

Example interaction with curl:

curl **-**vvv "http:**//**someuser:somepass@127.0.0.1:8080**/**routes"

**\*** About to connect() to 127.0.0.1 port 8080 (#0)

**\*** Trying 127.0.0.1*...*

**\*** connected

**\*** Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)

**\*** Server auth using Basic with user 'someuser'

**>** GET **/**routes HTTP**/**1.1

**>** Authorization: Basic c29tZXVzZXI6c29tZXBhc3M=

**>** User**-**Agent: curl**/**7.24.0 (x86\_64**-**apple**-**darwin12.0) libcurl**/**7.24.0 OpenSSL**/**0.9.8r zlib**/**1.2.5

**>** Host: 127.0.0.1:8080

**>** Accept: **\*/\***

**>**

**<** HTTP**/**1.1 200 OK

**<** Content**-**Type: application**/**json

**<** Date: Mon, 25 Mar 2013 20:31:27 GMT

**<** Transfer**-**Encoding: chunked

**<**

{"0295dd314aaf582f201e655cbd74ade5.cloudfoundry.me":["127.0.0.1:34567"],"03e316d6aa375d1dc1153700da5f1798.cloudfoundry.me":["127.0.0.1:34568"]}

# Schemata

This page should describe Schemata

# Stacks

A stack is a prebuilt file system, including an operating system, that supports running applications with certain characteristics. Any DEA can support exactly one stack. To stage or run a Linux app using MySQL, a DEA running the lucid64 stack must be available (and have free memory).

The scripts for building this stack (and later for building other stacks) reside in the [stacks](http://github.com/cloudfoundry/stacks) project.

## The lucid64 stack

Currently, Cloud Foundry supports one stack: lucid64, a Ubuntu 10.04 64-bit system containing a number of common programs and libraries including:

* MySQL
* PostgreSQL
* sqlite
* imagemagick
* git
* ruby 1.9.3

# User Account and Authentication (UAA) Server

The UAA is the identity management service for Cloud Foundry. It's primary role is as an OAuth2 provider, issuing tokens for client applications to use when they act on behalf of Cloud Foundry users. It can also authenticate users with their Cloud Foundry credentials, and can act as an SSO service using those credentials (or others). It has endpoints for managing user accounts and for registering OAuth2 clients, as well as various other management functions.

## Quick Start

If this works you are in business:

$ git clone git:**//**github.com**/**cloudfoundry**/**uaa.git

$ cd uaa

$ mvn install

Each module has a mvn tomcat:run target to run individually, or you could import them as projects into STS (use 2.8.0 or better if you can). The apps all work together the apps running on the same port (8080) as /uaa, /app and /api.

You will need Maven 3.0.4 or newer.

### Deploy to Cloud Foundry

You can also build the app and push it to Cloud Foundry, e.g.

$ mvn package install

$ vmc push myuaa **--**path uaa**/**target

(If you do that, choose a unique application id, not 'myuaa'.)

### Demo of command line usage on local server

First run the UAA server as described above:

$ cd uaa

$ mvn tomcat:run

Then start another terminal and from the project base directory, ask the login endpoint to tell you about the system:

$ curl **-**H "Accept: application**/**json" localhost:8080**/**uaa**/**login

{

"timestamp":"2012**-**03**-**28T18:25:49**+**0100",

"commit\_id":"111274e",

"prompts":{"username":["text","Username"],

"password":["password","Password"]

}

}

Then you can try logging in with the UAA ruby gem. Make sure you have ruby 1.9, then

$ gem install cf**-**uaac

$ uaac target http:**//**localhost:8080**/**uaa

$ uaac token get marissa koala

(or leave out the username / password to be prompted).

This authenticates and obtains an access token from the server using the OAuth2 implicit grant, similar to the approach intended for a client like VMC. The token is stored in ~/.uuac.yml, so dig into that file and pull out the access token for your vmc target (or use --verbose on the login command line above to see it logged to your console).

Then you can login as a resource server and retrieve the token details:

$ uaac target http:**//**localhost:8080**/**uaa

$ uaac token decode [token**-**value**-**from**-**above]

You should see your username and the client id of the original token grant on stdout, e.g.

exp: 1355348409

user\_name: marissa

scope: cloud\_controller.read openid password.write scim.userids tokens.read tokens.write

email: marissa@test.org

aud: scim tokens openid cloud\_controller password

jti: ea2fac72**-**3f51**-**4c8f**-**a7a6**-**5ffc117af542

user\_id: ba14fea0**-**9d87**-**4f0c**-**b59e**-**32aaa8eb1434

client\_id: vmc

### Demo of command line usage on cloudfoundry.com

The same command line example should work against a UAA running on cloudfoundry.com (except for the token decoding bit because you won't have the client secret). In this case, there is no need to run a local uaa server, so simply ask the external login endpoint to tell you about the system:

$ curl **-**H "Accept: application**/**json" uaa.cloudfoundry.com**/**login

{

"prompts":{"username":["text","Username"],

"password":["password","Password"]

}

}

You can then try logging in with the UAA ruby gem. Make sure you have ruby 1.9, then

$ gem install cf**-**uaac

$ uaac target uaa.cloudfoundry.com

$ uaac token get [yourusername] [yourpassword]

(or leave out the username / password to be prompted).

This authenticates and obtains an access token from the server using the OAuth2 implicit grant, the same as used by a client like VMC.

## Integration tests

With all apps deployed into a running server on port 8080 the tests will include integration tests (a check is done before each test that the app is running). You can deploy them in your IDE or using the command line with mvn tomcat:run and then run the tests as normal.

For individual modules, or for the whole project, you can also run integration tests and the server from the command line in one go with

$ mvn test **-**P integration

(This might require an initial mvn install from the parent directory to get the wars in your local repo first.)

To make the tests work in various environments you can modify the configuration of the server and the tests (e.g. the admin client) using a variety of mechanisms. The simplest is to provide additional Maven profiles on the command line, e.g.

$ (cd uaa; mvn test **-**P vcap)

will run the integration tests against a uaa server running in a local vcap, so for example the service URL is set to uaa.vcap.me (by default). There are several Maven profiles to play with, and they can be used to run the server, or the tests or both:

* local: runs the server on the ROOT context http://localhost:8080/
* vcap: also runs the server on the ROOT context and points the tests at uaa.vcap.me.
* devuaa: points the tests at http://devuaa.cloudfoundry.com (an instance of UAA deployed on cloudfoundry).

All these profiles set the CLOUD\_FOUNDRY\_CONFIG\_PATH to pick up a uaa.yml and (if appropriate) set the context root for running the server (see below for more detail on that).

### BVTs

There is a really simple cucumber feature spec (--tag @uaa) to verify that the UAA server is there. There is also a rake task to launch the integration tests from the uaasubmodule in vcap. Typical usage for a local (uaa.vcap.me) instance:

$ cd vcap**/**tests

$ rake bvt:run\_uaa

You can change the most common important settings with environment variables (see below), or with a custom uaa.yml. N.B. MAVEN\_OPTS cannot be used to set JVM system properties for the tests, but it can be used to set memory limits for the process etc.

### Custom YAML Configuration

To modify the runtime parameters you can provide a uaa.yml, e.g.

$ cat **>** **/**tmp**/**uaa.yml

uaa:

host: uaa.appcloud21.dev.mozycloud

test:

username: dev@cloudfoundry.org # defaults to vcap\_tester@vmware.com

password: changeme

email: dev@cloudfoundry.org

then from vcap-tests

$ CLOUD\_FOUNDRY\_CONFIG\_PATH=**/**tmp rake bvt:run\_uaa

or from uaa/uaa

$ CLOUD\_FOUNDRY\_CONFIG\_PATH=**/**tmp mvn test

The integration tests look for a Yaml file in the following locations (later entries override earlier ones), and the webapp does the same when it starts up so you can use the same config file for both:

classpath**:**uaa**.**yml

file**:**$**{**CLOUD\_FOUNDRY\_CONFIG\_PATH**}/**uaa**.**yml

file**:**$**{**UAA\_CONFIG\_FILE**}**

$**{**UAA\_CONFIG\_URL**}**

### Using Maven with Cloud Foundry or VCAP

To test against a vcap instance use the Maven profile vcap (it switches off some of the tests that create random client and user accounts):

$ (cd uaa; mvn test **-**P vcap)

To change the target server it should suffice to set VCAP\_BVT\_TARGET (the tests prefix it with uaa. to form the server url), e.g.

$ VCAP\_BVT\_TARGET=appcloud21.dev.mozycloud mvn test **-**P vcap

You can also override some of the other most important default settings using environment variables. The defaults as usual come from uaa.yml but tests will search first in an environment variable:

* UAA\_ADMIN\_CLIENT\_ID the client id for bootstrapping client registrations needed for the rest of the tests.
* UAA\_ADMIN\_CLIENT\_SECRET the client secret for bootstrapping client registrations

All other settings from uaa.yml can be overridden individually as system properties. Running in an IDE this is easy just using whatever features allow you to modify the JVM in test runs, but using Maven you have to use the argLine property to get settings passed onto the test JVM, e.g.

$ mvn **-**DargLine=**-**Duaa.test.username=foo test

will create an account with userName=foo for testing (instead using the default setting from uaa.yml).

If you prefer environment variables to system properties you can use a custom uaa.yml with placeholders for your environment variables, e.g.

uaa**:**

test**:**

username**:** $**{**UAA\_TEST\_USERNAME**:**marissa**}**

will look for an environment variable (or system property) UAA\_TEST\_USERNAME before defaulting to marissa. This is the trick used to expose UAA\_ADMIN\_CLIENT\_SECRET etc. in the standard configuration.

### Using Maven with to test with postgresql or mysql

The default uaa unit tests (mvn test) use hsqldb.

To run the unit tests using postgresql:

$ SPRING\_PROFILES\_ACTIVE=test,postgresql CLOUD\_FOUNDRY\_CONFIG\_PATH=src**/**test**/**resources**/**test**/**profiles**/**postgresql mvn test

To run the unit tests using mysql:

$ SPRING\_PROFILES\_ACTIVE=test,mysql CLOUD\_FOUNDRY\_CONFIG\_PATH=src**/**test**/**resources**/**test**/**profiles**/**mysql mvn test

The database configuration for the common and scim modules is located at: common/src/test/resources/(mysql|postgresql).properties scim/src/test/resources/(mysql|postgresql).properties

## Inventory

There are actually several projects here, the main uaa server application and some samples:

1. common is a module containing a JAR with all the business logic. It is used in the webapps below.
2. uaa is the actual UAA server
3. gem is a ruby gem (cf-uaa-client) for interacting with the UAA server
4. api (sample) is an OAuth2 resource service which returns a mock list of deployed apps
5. app (sample) is a user application that uses both of the above
6. login (sample) is an application that performs authentication for the UAA acting as a back end service

In CloudFoundry terms

* uaa provides an authentication service plus authorized delegation for back-end services and apps (by issuing OAuth2 access tokens).
* api is a service that provides resources that other applications may wish to access on behalf of the resource owner (the end user).
* app is a webapp that needs single sign on and access to the api service on behalf of users.
* login is where Cloud Foundry administrators set up their authentication sources, e.g. LDAP/AD, SAML, OpenID (Google etc.) or social. The cloudfoundry.com platform uses a different implementation of the [login server](https://github.com/cloudfoundry/login-server).

## UAA Server

The authentication service is uaa. It's a plain Spring MVC webapp. Deploy as normal in Tomcat or your container of choice, or execute mvn tomcat:run to run it directly from uaa directory in the source tree (make sure the common jar is installed first using mvn install from the common subdirectory or from the top level directory). When running with maven it listens on port 8080.

The UAA Server supports the APIs defined in the UAA-APIs document. To summarise:

1. The OAuth2 /authorize and /token endpoints
2. A /login\_info endpoint to allow querying for required login prompts
3. A /check\_token endpoint, to allow resource servers to obtain information about an access token submitted by an OAuth2 client.
4. SCIM user provisioning endpoint
5. OpenID connect endpoints to support authentication /userinfo and /check\_id (todo). Implemented roughly enough to get it working (so /app authenticates here), but not to meet the spec.

Authentication can be performed by command line clients by submitting credentials directly to the /authorize endpoint (as described in UAA-API doc). There is anImplicitAccessTokenProvider in Spring Security OAuth that can do the heavy lifting if your client is Java.

By default uaa will launch with a context root /uaa. There is a Maven profile local to launch with context root /, and another called vcap to launch at / with a postgresql backend.

### Configuration

There is a uaa.yml in the application which provides defaults to the placeholders in the Spring XML. Wherever you see ${placeholder.name} in the XML there is an opportunity to override it either by providing a System property (-D to JVM) with the same name, or a custom uaa.yml (as described above).

All passwords and client secrets in the config files are plain text, but they will be inserted into the UAA database encrypted with BCrypt.

### User Account Data

The default is to use an in-memory RDBMS user store that is pre-populated with a single test users: marissa has password koala.

To use Postgresql for user data, activate one of the Spring profiles hsqldb or postgresql.

The active profiles can be configured in uaa.yml using

spring\_profiles**:** postgresql

or by passing the spring.profiles.active parameter to the JVM. For, example to run with an embedded HSQL database:

mvn **-**Dspring.profiles.active=hsqldb tomcat:run

Or to use PostgreSQL instead of HSQL:

mvn **-**Dspring.profiles.active=postgresql tomcat:run

To bootstrap a microcloud type environment you need an admin client. For this there is a database initializer component that inserts an admin client. If the default profile is active (i.e. not postgresql) there is also a vmc client so that the gem login works out of the box. You can override the default settings and add additional clients inuaa.yml:

oauth**:**

clients**:**

admin**:**

authorized**-**grant**-**types**:** client\_credentials

scope**:** read**,**write**,**password

authorities**:** ROLE\_CLIENT**,**ROLE\_ADIN

id**:** admin

secret**:** adminclientsecret

resource**-**ids**:** clients

The admin client can be used to create additional clients (but not to do anything much else). A client with read/write access to the scim resource will be needed to create user accounts. The integration tests take care of this automatically, inserting client and user accounts as necessary to make the tests work.

## The API Application

An example resource server. It hosts a service which returns a list of mock applications under /apps.

Run it using mvn tomcat:run from the api directory (once all other tomcat processes have been shutdown). This will deploy the app to a Tomcat manager on port 8080.

## The App Application

This is a user interface app (primarily aimed at browsers) that uses OpenId Connect for authentication (i.e. SSO) and OAuth2 for access grants. It authenticates with the Auth service, and then accesses resources in the API service. Run it with mvn tomcat:run from the app directory (once all other tomcat processes have been shutdown).

The application can operate in multiple different profiles according to the location (and presence) of the UAA server and the Login application. By default it will look for a UAA on localhost:8080/uaa, but you can change this by setting an environment variable (or System property) called UAA\_PROFILE. In the application source code (src/main/resources) you will find multiple properties files pre-configured with different likely locations for those servers. They are all in the form application-<UAA\_PROFILE>.properties and the naming convention adopted is that the UAA\_PROFILE is local for the localhost deployment, vcap for a vcap.me deployment, stagingfor a staging deployment (inside VMware VPN), etc. The profile names are double barrelled (e.g. local-vcap when the login server is in a different location than the UAA server).

### Use Cases

1. See all apps
2. GET **/**app**/**apps

browser is redirected through a series of authentication and access grant steps (which could be slimmed down to implicit steps not requiring user at some point), and then the photos are shown.

1. See the currently logged in user details, a bag of attributes grabbed from the open id provider
2. GET **/**app

## The Login Application

A user interface for authentication. The UAA can also authenticate user accounts, but only if it manages them itself, and it only provides a basic UI. The Login app can be branded and customized for non-native authentication and for more complicate UI flows, like user registration and password reset.

The login application is actually itself an OAuth2 endpoint provider, but delegates those features to the UAA server. Configuration for the login application therefore consists of locating the UAA through its OAuth2 endpoint URLs, and registering the login application itself as a client of the UAA. There is a login.yml for the UAA locations, e.g. for a local vcap instance:

uaa**:**

url**:** http**://**uaa**.**vcap**.**me

token**:**

url**:** http**://**uaa**.**vcap**.**me/oauth/token

login**:**

url**:** http**://**uaa**.**vcap**.**me**/**login**.**do

and there is an environment variable (or Java System property), LOGIN\_SECRET for the client secret that the app uses when it authenticates itself with the UAA. The Login app is registered by default in the UAA only if there are no active Spring profiles (so not at all in vcap). In the UAA you can find the registration in the oauth-clients.xmlconfig file. Here's a summary:

id**:** login

secret**:** loginsecret

authorized**-**grant**-**types**:** client\_credentials

authorities**:** ROLE\_LOGIN

resource**-**ids**:** oauth

### Use Cases

1. Authenticate
2. GET **/**login

The sample app presents a form login interface for the backend UAA, and also an OpenID widget so the user can authenticate using Google etc. credentials.

1. Approve OAuth2 token grant
2. GET **/**oauth**/**authorize?client\_id=app**&**response\_type=code*...*

Standard OAuth2 Authorization Endpoint. Client credentials and all other features are handled by the UAA in the back end, and the login application is used to render the UI (seeaccess\_confirmation.jsp).

1. Obtain access token
2. POST **/**oauth**/**token

Standard OAuth2 Authorization Endpoint passed through to the UAA.

# Warden

Manages isolated, ephemeral, and resource controlled environments.

The project's primary goal is to provide a simple API for managing isolated environments. These isolated environments – or containers – can be limited in terms of CPU usage, memory usage, disk usage, and network access. As of writing, the only supported OS is Linux.

## Components

This repository contains the following components:

* warden – server
* warden-protocol – protocol definition, used by both the server and clients
* warden-client – client (Ruby)
* em-warden-client – client (Ruby's EventMachine)

## Getting Started

This short guide assumes Ruby 1.9 and Bundler are already available. Ensure that Ruby 1.9 has GNU readline library support through the package: 'libreadline-dev' and zlib support through the 'zlib1g-dev' package.

#### Install the right kernel

If you are running Ubuntu 10.04 (Lucid), make sure the backported Natty kernel is installed. After installing, reboot the system before continuing.

sudo apt**-**get install **-**y linux**-**image**-**generic**-**lts**-**backport**-**natty

#### Install dependencies

sudo apt**-**get install **-**y build**-**essential debootstrap

#### Setup Warden

Run the setup routine, which compiles the C code bundled with Warden and sets up the base file system for Linux containers.

sudo bundle exec rake setup[config**/**linux.yml]

**NOTE**: If sudo complains that bundle cannot be found, try sudo env PATH=$PATH to pass your current PATH to the sudo environment.

The setup routine sets up the file system for the containers at the directory path specified under the key: server -> containerrootfspath in the config file: config/linux.yml.

#### Run Warden

sudo bundle exec rake warden:start[config**/**linux.yml]

#### Interact with Warden

bundle exec bin**/**warden

## Implementation for Linux

Isolation is achieved by namespacing kernel resources that would otherwise be shared. The intended level of isolation is set such that multiple containers present on the same host should not be aware of each others presence. This means that these containers are given (among others) their own PID (Process ID) namespace, network namespace, and mount namespace.

Resource control is done by using [Control Groups](http://kernel.org/doc/Documentation/cgroups/cgroups.txt). Every container is placed in its own control group, where it is configured to use an equal slice of CPU compared to other containers, and the maximum amount of memory it may use.

The following sections give a brief summary of the techniques used to implement the Linux backend for Warden. A more detailed description can be found in theroot/linux directory of this repository.

### Networking

Every container is assigned a network interface which is one side of a virtual ethernet pair created on the host. The other side of the virtual ethernet pair is only visible on the host (from the root namespace). The pair is configured to use IPs in a small and static subnet. Traffic from and to the container can be forwarded using NAT. Additionally, all traffic can be filtered and shaped as needed, using readily available tools such as iptables.

### Filesystem

Every container gets a private root filesystem. This filesystem is created by stacking a read-only filesytem and a read-write filesystem. This is implemented by using aufson Ubuntu versions from 10.04 up to 11.10, and overlayfs on Ubuntu 12.04.

The read-only filesystem contains the minimal set of Ubuntu packages and Warden-specific modifications common to all containers. The read-write filesystem stores files overriding container-specific settings when necessary. Because all writes are applied to the read-write filesystem, containers can share the same read-only base filesystem.

The read-write filesystem is created by formatting a large sparse file. Because the size of this file is fixed, the filesystem that it contains cannot grow beyond this initial size.

### Difference with LXC

The Linux Containers or LXC project has goals that are similar to those of Warden; isolation and resource control. They both use the same Linux kernel primitives to achieve their goals. In fact, early versions of Warden even **used LXC**.

The major difference between the two projects is that LXC is explicitly tied to Linux, where Warden backends can be implemented for any operating system that implements some way of isolating environments. It is a daemon that manages containers and can be controlled via a simple API rather than a set of tools that are individually executed.

While the Linux backend for Warden was initially implemented with LXC, the current version no longer depends on it. During development, we found that running LXC out of the box is a very opaque and static process. There is little control over when different parts of the container start process are executed, and how they relate to each other. Because Warden relies on a very small subset of the functionality that LXC offers, we decided to create a tool that only implements the functionality we need in under 1k LOC of C code. This tool executes preconfigured hooks at different stages of the container start process, such that required resources can be set up without worrying about concurrency issues. These hooks make the start process more transparent, allowing for easier debugging when parts of this process are not working as expected.

## Container Lifecycle

The entire lifecyle of containers is managed by Warden. The API allows users to create, configure, use, and destroy containers. Additionally, it can automatically clean up unused containers when needed.

### Create

Every container is identified by its handle, which is returned by Warden upon creating it. It is a hexadecimal representation of the IP address that is allocated for the container. Regardless of whether the backend providing the container functionality supports networking or not, an IP address will be allocated by Warden to identify a container.

When a container was created and its handle was returned to the caller, it is immediately ready for use. All resources will be allocated, the necessary processes will be started and all firewalling tables will have been updated.

If Warden is configured to clean up containers after activity, it will use the number of connections that have referenced the container as a metric to determine inactivity. If the number of connections referencing the container drops to zero, the container will automatically be destroyed after a preconfigured interval. If in the mean time the container is referenced again, this timer is cancelled.

### Use

The container can be used by running arbitrary scripts, copying files in and out, modifying firewall rules and modifying resource limits. A complete list of operations is discussed under “Interface”.

### Destroy

When a container is destroyed – either per user request, or automatically after being idle – Warden first kills all unprivileged processes running inside the container. These processes first receive a TERM signal followed by a KILL if they haven't exited after a couple of seconds. When these processes have terminated, the root of the container's process tree is sent a KILL. Once all resources the container used have been released, its files are removed and it is considered destroyed.

## Networking

## Interface

Warden uses a line-based JSON protocol to communicate with its clients, and does so over a Unix socket which is located at /tmp/warden.sock by default. Every command invocation is formatted as a JSON array, where the first element is the command name and subsequent elements can be any JSON object. The commands it responds to are as follows:

**create [CONFIG]**

Creates a new container.

Returns the handle of the container which is used to identify it.

The optional CONFIG parameter is a hash that specifies configuration options used during container creation. The supported options are:

bind\_mounts

If supplied, this specifies a set of paths to be bind mounted inside the container. The value must be an array. The elements in this array specify the bind mounts to execute, and are executed in order. Every element must be of the form:

**[**

# Path **in** the host filesystem

"/host/path",

# Path **in** the container

"/path/in/container",

# Optional hash **with** options. The `mode` key specifies whether the bind

# mount should be remounted as `ro` (read-only) **or** `rw` (read-write).

{

"mode" **=>** "ro|rw"

}

**]**

grace\_time

If specified, this setting overrides the default time of a container not being referenced by any client until it is destroyed. The value can either be the number of seconds as floating point number or integer, or the null value to completely disable the grace time.

disk\_size\_mb

If specified, this setting overrides the default size of the container's scratch filesystem. The value is expected to be an integer number.

**spawn HANDLE SCRIPT [OPTS]**

Run the script SCRIPT in the container identified by HANDLE.

Returns a job identifier that can be used to reap its exit status at some point in the future. Also, the connection that issued the command may go away and reconnect later while still being able to reap the job.

The optional OPTS parameter is a hash that specifies options modifying the command being run. The supported options are:

privileged

If true, this specifies that the script should be run as root.

**link HANDLE JOB\_ID**

Reap the script identified by JOB\_ID, running in the container identified by HANDLE.

Returns a 3-element tuple containing the integer exit status, a string containing its STDOUT and a string containing its STDERR. These elements may be null when they cannot be determined (e.g. the script couldn't be executed, was killed, etc.).

**stream HANDLE JOB\_ID**

Stream STDOUT and STDERR of scripts identified by JOB\_ID, running in the container identified by HANDLE.

Returns a 2-element tuple containing the type of stream viz. STDOUT or STDERR as the first element, and a chunk of the stream as the second element. Returns an empty tuple when no more data is available in the stream.

**limit HANDLE (mem) [VALUE]**

Set or get resource limits for the container identified by HANDLE.

The following resources can be limited:

* The memory limit is specified in number of bytes. It is enforced using the control group associated with the container. When a container exceeds this limit, one or more of its processes will be killed by the kernel. Additionally, the Warden will be notified that an OOM happened and it subsequently tears down the container.

**net HANDLE in**

Forward a port on the external interface of the host to the container identified by HANDLE.

Returns the port number that is mapped to the container. This port number is the same on the inside of the container.

**net HANDLE out ADDRESS[/MASK][:PORT]**

Allow traffic from the container identified by HANDLE to the network address specified by ADDRESS. Additionally, the address may be masked to allow a network of addresses, and a port to only allow traffic to a specific port.

Returns ok.

**copy HANDLE in SRC\_PATH DST\_PATH**

Copy the contents at SRC\_PATH on the host to DST\_PATH in the container identified by HANDLE.

Returns ok.

File permissions and symbolic links will be preserved, while hardlinks will be materialized. If SRC\_PATH contains a trailing / only the contents of the directory will be copied. Otherwise, the outermost directory, along with its contents, will be copied. The unprivileged user will be the owner of the files in the container.

**copy HANDLE out SRC\_PATH DST\_PATH [OWNER]**

Copy the contents at SRC\_PATH in the container identified by HANDLE to DST\_PATH on the host.

Returns ok.

Its semantics are identical to copy HANDLE in except in respect to file ownership. By default, the files on the host will be owned by root. If the OWNER argument is supplied (in the form of USER:GROUP), files on the host will be chowned to this user/group after the copy has completed.

**stop HANDLE**

Stop processes running inside the container identified by HANDLE.

Returns ok.

Because all processes are taken down, unfinished scripts will likely terminate without an exit status being available.

**destroy HANDLE**

Stop processes and destroy all resources associated with the container identified HANDLE.

Returns ok.

Because everything related to the container is destroyed, artifacts from running an earlier script should be copied out before calling destroy.

## Configuration

Warden can be configured by passing a configuration file when it is started. An example configuration is located at config/linux.yml in the repository.

## System prerequisites

Warden runs on Ubuntu 10.04 and higher.

A backported kernel needs to be installed on 10.04. This kernel is available as linux-image-server-lts-backport-natty (substitute server for generic if you are running Warden on a desktop variant of Ubuntu 10.04).

Other dependencies are:

* build-essential (for compiling Warden's C bits)
* debootstrap (for bootstrapping the container's base filesystem)
* quota (for managing file system quotas)

Further bootstrapping of Warden can be done by running rake setup.

## Hacking

The included tests create and destroy real containers, so require system prerequisites to be in place. They need to be run as root if the backend to be tested requires it.

See root/<backend>/README.md for backend-specific information.

# Services

The Cloud Foundry core team have developed the APIs between services and the Cloud Controller, as well as several services which can be installed with Cloud Foundry. These services have been implemented with a common architecture. This architecture uses gateways and nodes.

Gateways advertise service offerings to the cloud controller and receive calls from cloud controller for four functions; create, delete, bind, and unbind. Gateways pass these calls to the service nodes.

Nodes are where the server processes for services are run. Nodes receive calls from gateways, execute the four functions, and return credentials to gateways.

### Adding custom Cloud Foundry Services

Many members of our open source community are developing custom services for their private Cloud Foundry instances. These built-in services are packaged and deployed with Cloud Foundry as jobs and typically support automated provisioning of service instances. We provide an simple example service to help learn how to develop your own Cloud Foundry services.

* [Echo Service](https://github.com/cloudfoundry/vcap-services-sample-release)

### Connecting external services to Cloud Foundry

Private Cloud Foundry operators frequently ask how applications pushed to their Cloud Foundry instances can bind with external services, such as Oracle, which they operate themselves.

* [Service Connector](http://docs.cloudfoundry.com/docs/running/architecture/services/service-connector.html)