调试

local ubuntu： test password

heroku： [tchen.data@gmail.com](mailto:tchen.data@gmail.com) chenteng

Gateway： 192.168.2.200

vCenter：192.168.2.103

NFS Server： 192.168.2.21

domain: vscf.com

ovf：192.168.2.31；linux：tempest password； ovf-console：admin password

cf-console: <http://console.vscf.com> 用户、密码是ovf：192.168.2.31上UAA--〉Administrator user credentials

boshcli，DNS：192.168.2.201

IP Range：192.168.2.105 - 192.168.2.199

Router：192.168.2.110

Oracle：192.168.2.221 administrator pass2013/

mono：192.168.2.91 test password

Pivotal CF ovf：

部署命令文件位置：/home/tempest-web/tempest/web/scripts/

部署文件位置：/var/tempest/workspaces/default/deployments/cf-34b5c04725ad5dea6d23.yml

DEA：vcap 54dddcdb881702439a06

实例位置：/var/vcap/data/warden/depot/179eink72kk/tmp/rootfs/app/.tomcat/logs/catalina.2013-10-24.log

# BOSH操作（Pivotal CF ovf虚拟机）

1.登陆Pivotal CF ovf虚拟机192.168.2.31 linux：tempest password

2.参考/home/tempest-web/tempest/web/scripts/目录里的bosh\_deploy.sh和common.sh，编写bosh\_debugstep.sh和bosh\_debug.sh。

bosh\_debug.sh：完成自动批处理部署的工作。通过输入‘./bosh\_debug.sh’直接运行批处理bosh指令的任务。

bosh\_debugstep.sh：完成加载bosh gem，建立bosh命令环境。先输入‘source bosh\_debugstep.sh’，然后使用’run\_bosh’命令执行bosh的操作（如run\_bosh vms等）。

3.common.sh：‘run\_via\_bosh\_gemfile bosh -n $@’这一行的‘-n’参数是表示不进行交互模式的bosh指令。

**验证内容：**

1. 完成硬件内网环境的部署，形成隔离的实验环境
2. 完成Cloud Foundry V2版本安装，验证图形安装过程
3. Cloud Foundry功能验证（稳定性、性能、扩展性、易用性）
4. 界面化Console控制台功能验证
5. 负载均衡、粘性session （E:\PAASstudy\hello-java\src\main\java\org\cloudfoundry\samples\HttpClientTest.java）



1. 动态扩展、
2. 故障迁移（ps -ea|grep java 查看dea上运行的java实例，判断dea上是否有java实例运行。关掉一台dea后，该dea上被停掉的应用实例会在其他dea上启动。）
3. 使用buildPack发布vseaf框架
4. cloudfoundry用户增加（通过cf命令行工具增加，增加后可以访问http://console.vscf.com；使用admin用户在<http://console.vscf.com中为新用户添加development>角色。

F:\cloudfoundry\v2\demo\buildpack>cf create-user --email test02@vscf.com --passw

ord password）

1. 使用buildpack发布dotNet web应用（基于mono、xsp）
2. 部署典型应用：部署胜利软件集成平台服务端应用（基于深化应用平台的权限控制、资源中心、用户CAS服务）

------------------ ------------------ ------------------ ------------------ ------------------ ------------------

1. 通过buildPack部署 Weblogic
2. 完成云盘Web端应用部署，完成文件存储后端程序部署，实现初步的云盘联动演示环境
3. 上传大文件超时（调整方案）、打开文件数过多（调整方案）

# github

#修改本地内容后，执行下面命令将内容更新到github服务器上。

git clone <https://github.com/tchenData/java-buildpack.git>

git clone https://github.com/tchenData/.net-buildpack.git

#git commit -m "add directory"

git commit -a -m "add directory"

git push

# 问题

1. Q：修改console的admin用户的密码后，再进行cf节点修改、重新部署cf节点操作时，会提示admin的用户密码无效。

A：将console的admin用户的密码改回初始密码后，可以重新部署cf节点。

1. Q：IE浏览器不能正常访问Pivotal CF的部署功能。

A：使用Chrome浏览器访问Pivotal CF的部署功能。

1. Q：checking for MONO\_MODULE... configure: error: Package requirements (mono >= 2.10.0) were not met:

A：export PKG\_CONFIG\_PATH=/usr/local/lib/pkgconfig

1. Q：open my ASP.NET web application created in VS2010 and targeted to Framework 4.0, I’m getting next error: Unrecognized attribute 'targetFramework'.

A：The version of xsp targeting .NET 4.0 is called "xsp4".

1. Q：Ubuntu上运行sqlplus出现“libaio.so.1找不到”的错误提示。

A：sudo aptitude install libaio1

1. Q：在github不fork一个buildpack项目，而是新定义一个buildpack项目，这个项目无法在v2 cloudfoundry中使用。
2. Q：dea有时出现运行状态异常情况

A： 使用bosh cck命令对vm进行修复，然后在pivotal cf ovf（192.168.2.31）中重新update cf。

1. 在mono上运行云盘时，需要在ubuntu上设置 libgdiplus、xsp等。参见‘mono--〉安装’
2. Q：cf上部署weblogic时，出现

A：- name: dea\_next

template: dea\_next

instances: 1

resource\_pool: infrastructure

networks:

- name: default

properties:

disk\_quota\_enabled: false

# cloudfoundry java-buildpack

Expanding Tomcat to .tomcat Command 'tar xzf /tmp/cache/http:%2F%2F192.16

8.2.103:8001%2Fbuildpack%2Ftomcat-7.0.23.tar.gz.cached -C /tmp/staged/app/.tomca

t --strip 1 --exclude webapps --exclude conf/server.xml --exclude conf/context.x

ml 2>&1' has failed

# mono

apt-get install gcc g++ bison gettext make

cd /usr/local/src

wget http://download.mono-project.com/sources/mono/mono-2.10.8.tar.gz

tar zxvf mono-2.10.8.tar.gz

cd mono-2.10.8

./configure --prefix=/usr/local

make && make install

apt-get install pre-config

使用xsp 2.10版本，低版本的xsp 有Mono.data sqllite dll丢失问题。

$ cd /src

$ wget [[http://pkgs.fedoraproject.org/repo/pkgs/xsp/xsp-2.10.tar.bz2](http://pkgs.fedoraproject.org/repo/pkgs/xsp/xsp-2.4.tar.bz2)](http://ftp.novell.com/pub/mono/sources/xsp/xsp-2.4.tar.bz2)

$ tar -xvf xsp-2.10.tar.bz2

$ cd xsp-2.4/

$ ./configure --prefix=/usr/local; make; make install

$ cd /usr/local/lib/xsp/test $ xsp2

Open your browser and point it to http://localhost:8080 (or your domain/ip if you're doing this remotely). You should see the XSP test page.

编写asp页面时，如果没有将html tag写完整，将无法运行asp页面。如：没有写</html>.

gmcs -r:System.Data.OracleClient oracle.cs

mono oracle.exe

## 安装

**Installing Mono**

Open a terminal window if you're using the Ubuntu GUI or log on using SSH if you're accessing a remote server.

All these instructions assume that you have root privileges, so if you're not logged in as root, enter the following and enter you password when prompted.

$ sudo bash

First you need to remove the version of Mono that is pre-installed with Ubuntu.

$ apt-get remove mono-common

That will remove all the extra packages that are installed with mono as they all depend on it.Be aware that it will also remove applications like f-spot and beagle, which will have to be re-installed after the mono upgrade if you wish to use them.

Next, create a folder for the source code you're about to compile:

$ mkdir /src

Then change to the src directory:

$ cd /src

Make sure that your /etc/apt/sources.list file has the universe and multiverse repositories included. If you installation didn't have any internet connectivity, there may be a chance that the installer disabled those repositories due to being unable to verify them. If that is the case, do the following:

$ nano /etc/apt/sources.list

Remove the # characters from the universe and multiverse entries and save the file (a [Nano Basics Guide](http://www.gentoo.org/doc/en/nano-basics-guide.xml) is available for people new to nano). I personally just enable everything.

Refresh the apt-get database.

$ apt-get update

You now need to install some Mono build dependencies (and some optional enhancements):

$ apt-get install build-essential pkg-config libglib2.0-dev bison libcairo2-dev libungif4-dev libjpeg62-dev libtiff4-dev gettext

（sudo apt-get -f install  强制安装）

Download libgdiplus:

$ wget http://ftp.novell.com/pub/mono/sources/libgdiplus/libgdiplus-2.4.tar.bz2 $ tar -xvf libgdiplus-2.4.tar.bz2 $ cd libgdiplus-2.4/

Now we can compile libgdiplus and install it:

$ ./configure --prefix=/usr/local; make; make install

Now go make yourself a snack, especially if you're doing this on a VM :P

After the compilation finally finishes, we need to make sure the new packages are visible to the system:

$ sh -c "echo /usr/local/lib >> /etc/ld.so.conf" $ /sbin/ldconfig

Now change back to the /src folder and download the latest release of mono (about 17 MB):

$ cd /src $ wget http://ftp.novell.com/pub/mono/sources/mono/mono-2.4.tar.bz2

Extract the compressed file, change into the mono folder and compile:

$ tar -xvf mono-2.4.tar.bz2$ cd mono-2.4 $ ./configure --prefix=/usr/local; make; make install

You can go make yourself a gourmet meal at this point as the compilation process takes a seriously long time (ok, maybe not that long, but us generation X/Y types don't really have the patience for C++ compilation).

Add mono to the bash path:

$ nano ~/.bashrc

And add the following lines at the end:

PATH=/usr/local/bin:$PATH LD\_LIBRARY\_PATH=/usr/local/lib/:$LD\_LIBRARY\_PATH PKG\_CONFIG\_PATH=/usr/local/lib/pkgconfig:$PKG\_CONFIG\_PATH

In order for the changes to take effect, you need to either close and open your terminal again, or simply type:

$ bash

to start a new instance of bash command line.

To check your new mono version, type:

$ mono -V

You should see the following:

Mono JIT compiler version 2.4 (tarball Wed Apr 1 04:49:16 CDT 2009)Copyright (C) 2002-2008 Novell, Inc and Contributors. www.mono-project.com TLS: \_\_thread GC: Included Boehm (with typed GC) SIGSEGV: altstack Notifications: epoll Architecture: amd64 Disabled: none

Congratulations, you have a working copy of the latest Mono released installed on your Ubuntu machine!

**Installing XSP**

XSP is the Mono web server that Apache delegates to (via mod\_mono) when serving ASP.NET pages on Linux. In order to run ASP.NET pages you will need to compile and build the XSP web server.

Change to your /src directory, download the XSP sources, unzip them and compile:

$ cd /src $ wget http://ftp.novell.com/pub/mono/sources/xsp/xsp-2.4.tar.bz2$ tar -xvf xsp-2.4.tar.bz2 $ cd xsp-2.4/ $ ./configure --prefix=/usr/local; make; make install

Once the compilation is done, we can test the XSP server. Change to the XSP test installation folder and run the XSP server:

$ cd /usr/local/lib/xsp/test $ xsp2

You should see something like the following:

xsp2Listening on address: 0.0.0.0Root directory: /rootListening on port: 8080 (non-secure)Hit Return to stop the server.

Open your browser and point it to http://localhost:8080 (or your domain/ip if you're doing this remotely). You should see the XSP test page.

# WebLogic

1. 使用silent.xml安装weblogic出现针对silent.xml文件的错误提示时，将silent.xml文件中的<?xml version="1.0" encoding="UTF-8"?>这一行内容至于第一行。

java -Xmx1024m -jar wls1034\_generic.jar -mode=silent -silent\_xml=silent.xml -log=mylog.txt -log\_priority=debug

1. 注意silent.xml文件中的COMPONENT\_PATHS的值拼写正确性。
2. sh /home/test/Oracle/Middleware\_Home/wlserver\_10.34/common/bin/config.sh -mode=silent -silent\_script=configDomains.xml -log=configDomains.txt

# 如何手动重置 10 天激活宽限期

win 2012 datacenter 评估版过期后，会每小时关机一次。解决办法如下：

　　在最初的 10 天激活期限快要到期时，您可以运行 Slmgr.vbs 脚本将激活期限重置为 10 天。要进行操作，请执行以下步骤：

　　1. 单击“开始”，然后单击“命令提示符”。

　　2. 键入 slmgr.vbs -dli，然后按 Enter 以检查激活期限的当前状态。

　　3. 要重置激活期限，请键入 slmgr.vbs /rearm，然后按 Enter。

　　4. 重新启动计算机。

　　这样可将激活期限重置为 10 天，并且此命令至多可使用 5 次。在此之后，您将需要激活或重新安装该评估版软件。

# DNS Wildcard Configuration

You are viewing a wiki page. You are welcome to [**join the group and then edit it**](https://groups.drupal.org/og/subscribe/4701?destination=node%2F16862).[**Be bold!**](http://en.wikipedia.org/wiki/WP:BB)

Posted by [spiderman](https://groups.drupal.org/user/202) on November 18, 2008 at 5:38pm  
Last updated by [mariomaric](https://groups.drupal.org/user/21884) on Sun, 2011-10-09 02:32

**IMPORTANT NOTE:** Quick and efficient DNS Wildcard Configuration for local development @ Ubuntu/Debian is contained in [VII. Alternative dns server, dnsmasq setup](https://groups.drupal.org/node/16862#dnsmasq) step.

A wildcard DNS setup lets you automatically have subdomains for a given domain. For example, say you own the domain widgets.com and you want to setup an unlimited number of subdomains like dev.widgets.com, test.widgets.com, customers.widgets.com, etc.... Typically you would have to set these all up individually. A wildcard DNS can let you bypass a lot of configuration. In a development environment it can let you setup any number of test/development sites very quickly and easily. Drupal developers in particular can leverage Drupal's multisite installation feature to setup lots of sites for development or production very quickly.

Content:

1. [Ubuntu/Debian](https://groups.drupal.org/node/16862#debian)
2. [OS X Snow Leopard](https://groups.drupal.org/node/16862#osx)

### Setting up a Wildcard DNS configuration on Ubuntu/Debian

#### Overview of Steps on Ubuntu/Debian

I. Install Bind 9  
II. Add a zone to /etc/bind/named.conf.local. Check the syntax of named.conf.local with named-checkconf  
III. Add a zone file & Check the syntax of your zone files for errors with named-checkzone  
IV. Edit /etc/resolv.conf  
V. Start up Bind  
VI. Check setup with dig  
VII. Troubleshoot if needed  
VIII. Alternative dns server, dnsmasq setup (for local development, skip steps I., II. and III.)

#### I. Install Bind 9

$ sudo apt-get install bind9

#### II. Edit /etc/bind/named.conf.local to add a zone.

We need to add a zone to /etc/bind/named.conf.local. A zone is a record for a domain. In this case we'll use a fake domain for development purposes. Our domain can be called anything. Let's call it mydev. (I was really tempted to call the domain bullshit - would have been great for screencasts). So at a shell prompt do:

$ sudo nano /etc/bind/named.conf.local

Add this to the end of the file:

zone "mydev" {  
type master;  
   file "/etc/bind/db.mydev";  
};

Save the file with Control - O and enter. Exit from nano with Control X.

Check the syntax of the file as follows:

$ named-checkconf /etc/bind/named.conf.local

If the file is syntactically correct the shell prompt will return nothing. If there is an error - make sure you did not miss a quote or semicolon and recheck the file.

#### III. Add a zone file & Check the syntax of your zone files for errors with named-checkzone

$ sudo nano /etc/bind/db.mydev

Add this to the file replacing 10.0.2.15 with your IP address

mydev. 86400 IN SOA mydev. hostmaster.mydev. (  
               20091028 ; serial yyyy-mm-dd  
               10800; refresh every 15 min  
                3600; retry every hour  
                 3600000; expire after 1 month +  
                86400 ); min ttl of 1 day  
  IN NS mydev.   
          IN MX 10 mydev.  
        IN A 10.0.2.15  
\*.mydev. IN A 10.0.2.15

Save the file with Control - O and enter. Exit from nano with Control X. Check the syntax with the following:

$ named-checkzone mydev /etc/bind/db.mydev

If the syntax of the file is correct you should see something like:

zone mydev/IN: loaded serial 20091028  
OK

If not review the file for errors. In particular whenever referring to a domain, it must end in a . , thus mydev.

#### IV. Edit /etc/resolv.conf

The **resolv.conf** file tells applications like browsers where to look for DNS info. By default it may have your ISPs info. We need to tell it to check our local DNS server first then the ISP.

If you are receiving an IP address via [DHCP](http://en.wikipedia.org/wiki/Dhcp) (which is true in vast majority of cases) then you just need to uncomment one line in **/etc/dhcp3/dhclient.conf** file in order to include local DNS server in **/etc/resolv.conf** file.

Open file with nano editor:

sudo nano /etc/dhcp3/dhclient.conf

Look for the line:

#prepend domain-name-servers 127.0.0.1;

and set it as follows (e.g. delete **#** (hash) symbol):

prepend domain-name-servers 127.0.0.1;

Save changes and restart DHCP client:

sudo dhclient

In case you are not using DHCP, edit **/etc/resolv.conf** file and add the following line: nameserver 127.0.0.1, e.g.:

domain local-domain-name  
search local-domain-name  
nameserver 127.0.0.1  
nameserver x.x.x.x

It is important that inserted line is before nameserver x.x.x.x line.  
local-domain-name is, as name suggest, local domain and x.x.x.x is IP address of ISP nameserver.

#### V. Start up Bind (or Restart)

$ sudo /etc/init.d/bind9 restart

#### VI. Check setup with ping and dig

$  ping testing.mydev  
PING testing.mydev (10.0.2.15) 56(84) bytes of data.  
64 bytes from ubuntu.local (10.0.2.15): icmp\_seq=1 ttl=64 time=0.041 ms  
64 bytes from ubuntu.local (10.0.2.15): icmp\_seq=2 ttl=64 time=0.039 ms  
64 bytes from ubuntu.local (10.0.2.15): icmp\_seq=3 ttl=64 time=0.0

If ping works then you should be good to go. If not try dig and part V. To stop the ping action press [Ctrl]+[c].

#### VII. Troubleshoot

I found the section on Troubleshooting Bind in O'Reilly's "Linux System Administration" helpful. Fortunately, this part of the book is viewable on Google books as of this writing (10/29/2009): [http://books.google.com/books?id=-jYe2k1p5tIC&lpg=PP1&dq=Linux%20System%...](http://books.google.com/books?id=-jYe2k1p5tIC&lpg=PP1&dq=Linux%20System%20Administration&pg=PA66#v=onepage&q=&f=false) . See page 66

#### VIII. Alternative dns server, dnsmasq setup

[Dnsmasq](http://en.wikipedia.org/wiki/Dnsmasq) is a lightweight, easy to configure DNS forwarder and DHCP server.

Run following commands in order to install and set up dnsmasq (remove***sudo***if logged as root):

sudo apt-get install dnsmasq  
sudo echo "address=/mydev/127.0.0.1" > /etc/dnsmasq.d/mydev  
sudo /etc/init.d/dnsmasq restart

Finally, go through [IV. Edit /etc/resolv.conf](https://groups.drupal.org/node/16862#resolfconf) step.

### Getting a Wildcard DNS running on OSX Snow Leopard

Note: it took me several tries to get this working. If you are new to DNS be patient with yourself - you'll get it, but may take a few tries.

In this example I will concentrate on setting a development environment with OSX using wildcard DNS

### Overview of Steps

I. Edit /etc/named.conf to add a zone.  
II. Add a zone file at /var/named/  
III. Check the syntax of named.conf and your zone files for errors  
IV. Edit /etc/resolv.conf  
V. Set your computers network settings to use 127.0.0.1 as a name server  
VI. Start up Bind  
VII. Check setup with dig  
VIII. Reboot if needed

#### Before we start a few more notes.

I tend to use the nano text editor to edit Unix configuration files you could use Emacs, VI, Textmate, BBEdit or the editor of your choice.

Backup all these files we are editing so you can start over if you mess up. I didn't do this and it added more time to the project.  
For example to backup /etc/named.conf do:

$ sudo cp /etc/named.conf /etc/named.conf.bck

Last, most of the files we need to edit are owned by root so you will need to use sudo to edit these files. If you get tired of typing sudo you can become root by doing this:

$ sudo -s

Be careful when working as root or using sudo. You can mess up your system so make sure to backup. All example here are run as root.

#### I. Edit /etc/named.conf to add a zone

We need to edit named.conf to add our zone.

$ nano /etc/named.conf

I called my zone vmdev so I added this to named.conf

zone "vmdev" IN {  
        type master;  
        file "db.vmdev";  
};

I added this right before the zone 0.0.127.inaddr.apra and saved the file. So we told Bind to look in /var/named/db.vmdev for this zone.

#### II. Add a zone file at /var/named/

$ nano /var/named/db.vmdev  
vmdev. 7200    IN       SOA     vmdev. root.vmdev. (  
                                        2008031801 ;    Serial  
                                        15      ; Refresh every 15 minutes  
                                        3600    ; Retry every hour  
                                        3000000 ; Expire after a month+  
                                        86400 ) ; Minimum ttl of 1 day  
                IN      NS      vmdev.  
                IN      MX      10 vmdev.  
  
  
                IN      A       192.168.0.199  
\*.vmdev.        IN      A       192.168.0.199

You can just copy this but be sure to change 192.168.0.199 to you Mac's IP address

#### III. Check the syntax of named.conf and your zone files for errors

Run this to check your named.conf file:

$ named-checkconf /etc/named.conf

If it returns nothing, your named.conf file is at least syntactically correct. If there is an error, then well you have to diagnose and fix the error.

Now run this to check your zone file:

$ named-checkzone vmdev /var/named/db.vmdev

It should return something like this:

zone vmdev/IN: loaded serial 2008031801  
OK

If there are errors then diagnose and fix them.

#### IV. Edit /etc/resolv.conf

$ nano /etc/resolv.conf  
#  
# Mac OS X Notice  
#  
# This file is not used by the host name and address resolution  
# or the DNS query routing mechanisms used by most processes on  
# this Mac OS X system.  
#  
# This file is automatically generated.  
#  
domain vmdev.  
nameserver 127.0.0.1  
nameserver 192.168.0.1

#### V. Set your computers network settings to use 127.0.0.1 as a name server

Do this at System Preferences -> Network.  
You may want to use your ISPs Name server as the second name server

#### VI. Start up Bind

$ launchctl load -w /System/Library/LaunchDaemons/org.isc.named.plist

The -w option tells OSX to enable Bind at startup

#### VII. Check setup with dig

$ dig faker.vmdev

Should return something like this:

; <<>> DiG 9.6.0-APPLE-P2 <<>> faker.vmdev  
;; global options: +cmd  
;; Got answer:  
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 45640  
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 1  
  
;; QUESTION SECTION:  
;faker.vmdev.                   IN      A  
  
;; ANSWER SECTION:  
faker.vmdev.            7200    IN      A       192.168.0.199  
  
;; AUTHORITY SECTION:  
vmdev.                  7200    IN      NS      vmdev.  
  
;; ADDITIONAL SECTION:  
vmdev.                  7200    IN      A       192.168.0.199  
  
;; Query time: 0 msec  
;; SERVER: 127.0.0.1#53(127.0.0.1)  
;; WHEN: Wed Oct 14 19:28:56 2009  
;; MSG SIZE  rcvd: 75

The key here is status NOERROR; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 45640  
If you get an error then check previous steps.

Now try:

$ ping faker.vmdev

should return:

PING test.vmdev (192.168.0.199): 56 data bytes  
64 bytes from 192.168.0.199: icmp\_seq=0 ttl=64 time=0.059 ms  
64 bytes from 192.168.0.199: icmp\_seq=1 ttl=64 time=0.087 ms

but if it does not got to VIII.

#### VIII. Reboot if needed.

I needed to reboot to get everything to take. Whoila! Have a cookie or something.

Categories: [Documentation](https://groups.drupal.org/taxonomy/term/6181)

# tar.gz文件的处理

[7-ZIP](http://sparanoid.com/lab/7z/)这款开源软件完成此任务。

* 将文件打包成 tar 格式的压缩包
* 将打包好的 tar 压缩成 Gzip 格式，

Buildpacks

Cloud Foundry stages application using framework and and runtime-specific buildpacks. Heroku developed the buildpack approach, and made it available to the open source community.

* **Cloud Foundry Buildpacks** — When you push an application the runs in a runtime and framwork supported by a Cloud Foundry buildpack, the appropriate buildpack is automatically applied to the application. For information about the software resources each buildpack provides see:
  + [Java Buildpack](http://docs.cloudfoundry.com/docs/using/deploying-apps/java-buildpack.html)
  + [Ruby Buildpack](http://docs.cloudfoundry.com/docs/using/deploying-apps/ruby-buildpack.html)
  + [Node.js Buildpack](http://docs.cloudfoundry.com/docs/using/deploying-apps/node-buildpack.html)
* **External Buildpacks** — If you have an application that uses a language or framework that Cloud Foundry buildpacks do not support, there may be a third-party or community-developed buildpack that you can use. You can also customize an existing buildpack, or write your own.
  + [Cloud Foundry Commmunity Buildpacks](https://github.com/cloudfoundry-community/cf-docs-contrib/wiki/Buildpacks) — This page has links to buildpacks contributed by members of the Cloud Foundry Community.
  + [Heroku Third-Party Buildpacks](https://devcenter.heroku.com/articles/third-party-buildpacks) for a list of community-developed buildpacks. — This page has links to buildpacks developed for Heroku, which may (but have not been verified to) work with Cloud Foundry.
  + [Custom Buildpacks](http://docs.cloudfoundry.com/docs/using/deploying-apps/custom-buildpacks.html) — See this page for information about writing a custom buildpack.

To use a buildpack that is not built-in to Cloud Foundry, you specify the URL of the buildpack when you push an application, using the --buildpack qualifier.

About the Java Buildpack

Java Buildpack Documentation

For information about using, configuring, and extending the Cloud Foundry Java buildpack, see<https://github.com/cloudfoundry/java-buildpack>.

For information about the software installed by the Java buildpack, see the following section.

Software Installed by the Java Buildpack

The table below lists:

* **Name** — The name of the software installed by the Cloud Foundry Java buildpack, when appropriate.
* **Available Versions** — The versions of the software that are available from the buildpack. Note that the available versions may be dependent on the platform that the buildpack is run on.
* **Installed by Default** — The version of the software that is installed by default. See <https://github.com/cloudfoundry/java-buildpack/blob/master/docs/util-repositories.md#version-wildcards> for an explanation of the format used to indicate the version in the “Installed by Default” column.

This page was last updated on 19 September, 2013

| NAME | AVAILABLE VERSIONS | INSTALLED BY DEFAULT |
| --- | --- | --- |
| OpenJDK | lucid: [1.6.0\_21 -> 1.8.0\_M8](http://download.pivotal.io.s3.amazonaws.com/openjdk/lucid/x86_64/index.yml) mountainlion: [1.7.0\_04 -> 1.8.0\_M8](http://download.pivotal.io.s3.amazonaws.com/openjdk/mountainlion/x86_64/index.yml) precise: [1.7.0\_01 -> 1.8.0\_M8](http://download.pivotal.io.s3.amazonaws.com/openjdk/precise/x86_64/index.yml) | [1.7.0\_+](https://github.com/cloudfoundry/java-buildpack/blob/master/config/openjdk.yml) |
| Groovy | [1.5.0 -> 2.1.7](http://download.pivotal.io.s3.amazonaws.com/groovy/index.yml) | [2.1.+](https://github.com/cloudfoundry/java-buildpack/blob/master/config/groovy.yml) |
| Spring Boot CLI | [0.5.0\_M2 -> 0.5.0\_M4](http://download.pivotal.io.s3.amazonaws.com/spring-boot-cli/index.yml) | [0.5.0\_+](https://github.com/cloudfoundry/java-buildpack/blob/master/config/springbootcli.yml) |
| Tomcat | [6.0.0 -> 7.0.42](http://download.pivotal.io.s3.amazonaws.com/tomcat/index.yml) | [7.0.+](https://github.com/cloudfoundry/java-buildpack/blob/master/config/tomcat.yml) |
| Auto Reconfiguration | [0.6.8 -> 0.7.1](http://download.pivotal.io.s3.amazonaws.com/auto-reconfiguration/index.yml) | [0.+](https://github.com/cloudfoundry/java-buildpack/blob/master/config/springautoreconfiguration.yml) |
| Play JPA Plugin | [0.7.1](http://download.pivotal.io.s3.amazonaws.com/play-jpa-plugin/index.yml) | [0.+](https://github.com/cloudfoundry/java-buildpack/blob/master/config/playautoreconfiguration.yml) |
| New Relic | [2.11.0 -> 2.21.4](http://download.pivotal.io.s3.amazonaws.com/new-relic/index.yml) | [2.21.+](https://github.com/cloudfoundry/java-buildpack/blob/master/config/newrelic.yml) |

# Custom Buildpacks

Buildpacks are a convenient way of packaging framework and/or runtime support for your application. For example, by default Cloud Foundry does not support Python, or a Python framework like Django. Using a buildpack for Python and Django would allow you to add support for these at the deployment stage.

When an application written using one of these languages and frameworks is pushed, the required buildpack is automatically installed on the Cloud Foundry Droplet Execution Agent (DEA) where the application will run.

## Custom Buildpacks

The structure of a buildpack is straightforward. A buildpack repository contains three main scripts, situated in a folder named bin.

### bin/detect

The detect script is used to determine whether or not to apply the buildpack to an application. The script is called with one argument, the build directory for the application. It returns an exit code of 0 if the application can be supported by the buildpack. If the script does return 0, it should also print a framework name to STDOUT.

Shown below is an example detect script written in Ruby that checks for a Ruby application based on the existence of a Gemfile:

*#!/usr/bin/env ruby*

gemfile\_path **=** File**.join** ARGV**[**0**]**, "Gemfile"

**if** File**.exist?**(gemfile\_path)

puts "Ruby"

exit 0

**else**

exit 1

**end**

### bin/compile

The compile script builds the droplet that will be run by the DEA and will therefore contain all the components necessary to run the application.

The script is run with two arguments, the build directory for the application and the cache directory, which is a location the buildpack can use to store assets during the build process.

During execution of this script all output sent to STDOUT will be relayed via CF back to the end user. The generally accepted pattern for this is to break out this functionality in to a 'language\_pack'. A good example of this can be seen at <https://github.com/cloudfoundry/heroku-buildpack-ruby/blob/master/lib/language_pack/ruby.rb>

A simple compile script is shown below:

*#!/usr/bin/env ruby*

*#sync output*

$stdout**.sync** **=** **true**

build\_path **=** ARGV**[**0**]**

cache\_path **=** ARGV**[**1**]**

install\_ruby

**private**

**def install\_ruby**

puts "Installing Ruby"

*# !!! build tasks go here !!!*

*# download ruby to cache\_path*

*# install ruby*

**end**

### bin/release

The release script provides feedback metadata back to Cloud Foundry indicating how the application should be executed. The script is run with one argument, the build location of the application. The script must generate a YAML file in the following format:

config\_vars:

name: value

default\_process\_types:

web: commandLine

Where config\_vars is an optional set of environment variables that will be defined in the environment in which the application is executed. default\_process\_typesindicates the type of application being run and the command line used to start it. At this time only web type of applications are supported.

The following example shows what a Rack application's release script might return:

config\_vars:

RACK\_ENV: production

default\_process\_type:

web: bundle exec rackup config**.ru** **-**p $PORT

## Deploying With a Custom Buildpack

Once a custom buildpack has been created and pushed to a public git repository, the git URL can be passed via the cf command when pushing an application.

For example, for a buildpack that has been pushed to Github:

$ cf push my-new-app --buildpack=git://github.com/johndoe/my-buildpack.git

Alternatively, it is possible to use a private git repository (with https and username/password authentication) as follows:

$ cf push my-new-app --buildpack=https://username:password@github.com/johndoe/my-buildpack.git

The application will then be deployed to Cloud Foundry, and the buildpack will be cloned from the repository and applied to the application (provided that the detectscript returns 0).

# Getting Started with Java on Heroku

*Last Updated: 05 September 2013*

[**java**](https://devcenter.heroku.com/tags/java)

### Table of Contents

* [Prerequisites](https://devcenter.heroku.com/articles/getting-started-with-java#prerequisites)
* [Local workstation setup](https://devcenter.heroku.com/articles/getting-started-with-java#local-workstation-setup)
* [Write your app](https://devcenter.heroku.com/articles/getting-started-with-java#write-your-app)
* [Declare dependencies in pom.xml](https://devcenter.heroku.com/articles/getting-started-with-java#declare-dependencies-in-pom-xml)
* [Build and run your app locally](https://devcenter.heroku.com/articles/getting-started-with-java#build-and-run-your-app-locally)
* [Declare process types with Procfile](https://devcenter.heroku.com/articles/getting-started-with-java#declare-process-types-with-procfile)
* [Optionally Choose a JDK](https://devcenter.heroku.com/articles/getting-started-with-java#optionally-choose-a-jdk)
* [Store your app in Git](https://devcenter.heroku.com/articles/getting-started-with-java#store-your-app-in-git)
* [Deploy your application to Heroku](https://devcenter.heroku.com/articles/getting-started-with-java#deploy-your-application-to-heroku)
* [Visit your application](https://devcenter.heroku.com/articles/getting-started-with-java#visit-your-application)
* [Dyno sleeping and scaling](https://devcenter.heroku.com/articles/getting-started-with-java#dyno-sleeping-and-scaling)
* [View the logs](https://devcenter.heroku.com/articles/getting-started-with-java#view-the-logs)
* [Next steps](https://devcenter.heroku.com/articles/getting-started-with-java#next-steps)

This quickstart will get you going with Java and the [Jetty](http://eclipse.org/jetty/) embedded web server, deployed to Heroku. For general information on how to develop and architect apps for use on Heroku, see [Architecting Applications for Heroku](https://devcenter.heroku.com/articles/architecting-apps).

Sample code for the [**Java demo application**](https://github.com/heroku/devcenter-java) is available on GitHub. Edits and enhancements are welcome.

## [Prerequisites](https://devcenter.heroku.com/articles/getting-started-with-java#prerequisites)

* Basic Java knowledge, including an installed version of the JVM and[Maven 3](http://maven.apache.org/download.html).
* Your application must run on the [OpenJDK](http://openjdk.java.net/) version 6, or 7 (8 is also available in beta).
* A Heroku user account. [Signup is free and instant.](https://api.heroku.com/signup/devcenter)

## [Local workstation setup](https://devcenter.heroku.com/articles/getting-started-with-java#local-workstation-setup)

If you’re an Eclipse user you can skip these steps and get started inside of an Eclipse workspace. See [Getting Started with Heroku & Eclipse](https://devcenter.heroku.com/articles/getting-started-with-heroku-eclipse).

Install the [Heroku Toolbelt](https://toolbelt.heroku.com/) on your local workstation. This ensures that you have access to the [Heroku command-line client](https://devcenter.heroku.com/categories/command-line), Foreman, and the Git revision control system.

Once installed, you can use the heroku command from your command shell. Log in using the email address and password you used when creating your Heroku account:

$ heroku login

Enter your Heroku credentials.

Email: adam@example.com

Password:

Could not find an existing public key.

Would you like to generate one? [Yn]

Generating new SSH public key.

Uploading ssh public key /Users/adam/.ssh/id\_rsa.pub

Press enter at the prompt to upload your existing ssh key or create a new one, used for pushing code later on.

## [Write your app](https://devcenter.heroku.com/articles/getting-started-with-java#write-your-app)

You can run any Java application on Heroku that uses Maven as a build tool. As an example, we will write a web app using Jetty. Here is a basic servlet class that also contains a main method to start up the application:

### [src/main/java/HelloWorld.java](https://devcenter.heroku.com/articles/getting-started-with-java#src-main-java-helloworld-java)

import java.io.IOException;

import javax.servlet.ServletException;

import javax.servlet.http.\*;

import org.eclipse.jetty.server.Server;

import org.eclipse.jetty.servlet.\*;

public class HelloWorld extends HttpServlet {

@Override

protected void doGet(HttpServletRequest req, HttpServletResponse resp)

throws ServletException, IOException {

resp.getWriter().print("Hello from Java!\n");

}

public static void main(String[] args) throws Exception{

Server server = new Server(Integer.valueOf(System.getenv("PORT")));

ServletContextHandler context = new ServletContextHandler(ServletContextHandler.SESSIONS);

context.setContextPath("/");

server.setHandler(context);

context.addServlet(new ServletHolder(new HelloWorld()),"/\*");

server.start();

server.join();

}

}

## [Declare dependencies in pom.xml](https://devcenter.heroku.com/articles/getting-started-with-java#declare-dependencies-in-pom-xml)

Heroku recognizes Java apps by the existence of a pom.xml file. Here’s an example pom.xml for the Java/Jetty app we created above.

### [pom.xml](https://devcenter.heroku.com/articles/getting-started-with-java#pom-xml)

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<version>1.0-SNAPSHOT</version>

<artifactId>helloworld</artifactId>

<dependencies>

<dependency>

<groupId>org.eclipse.jetty</groupId>

<artifactId>jetty-servlet</artifactId>

<version>7.6.0.v20120127</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.5</version>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-dependency-plugin</artifactId>

<version>2.4</version>

<executions>

<execution>

<id>copy-dependencies</id>

<phase>package</phase>

<goals><goal>copy-dependencies</goal></goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

</project>

Prevent build artifacts from going into revision control by creating this file:

### [.gitignore](https://devcenter.heroku.com/articles/getting-started-with-java#gitignore)

target

## [Build and run your app locally](https://devcenter.heroku.com/articles/getting-started-with-java#build-and-run-your-app-locally)

Build your app locally:

$ mvn package

As part of the build, Maven gathers dependencies and copies them into the directory target/dependency. Start you app locally by setting the PORT environment variable and running Java with all dependencies on the classpath:

On Mac & Linux:

$ export PORT=5000

$ java -cp target/classes:"target/dependency/\*" HelloWorld

(double quotes needed to prevent expansion of \*)

On Windows:

$ set PORT=5000

$ java -cp target\classes;"target\dependency\\*" HelloWorld

You should now see something similar to:

2012-01-31 15:51:21.811:INFO:oejs.Server:jetty-7.6.0.v20120127

2012-01-31 15:51:21.931:INFO:oejsh.ContextHandler:started o.e.j.s.ServletContextHandler{/,null}

2012-01-31 15:51:21.971:INFO:oejs.AbstractConnector:Started SelectChannelConnector@0.0.0.0:5000

Open the app in your browser: [http://localhost:5000](http://localhost:5000/)

## [Declare process types with Procfile](https://devcenter.heroku.com/articles/getting-started-with-java#declare-process-types-with-procfile)

Use a [Procfile](https://devcenter.heroku.com/articles/procfile), a text file in the root directory of your application, to explicitly declare what command should be executed to start a web [dyno](https://devcenter.heroku.com/articles/dynos). In this case, we need to simply run the HelloWorld class.

Here’s what the Procfile looks like:

web: java -cp target/classes:target/dependency/\* HelloWorld

(note: no double quotes needed in Procfile)

## [Optionally Choose a JDK](https://devcenter.heroku.com/articles/getting-started-with-java#optionally-choose-a-jdk)

By default, OpenJDK 1.6 is installed with your app. However, you can choose to use a newer JDK by specifying java.runtime.version=1.7 in a system.properties file that you place in the root directory of your application.

Here’s what a system.properties file looks like:

java.runtime.version=1.7

You can specify 1.6, 1.7, or 1.8 (1.8 is in beta) for Java 6, 7, or 8 (with lambdas), respectively.

## [Store your app in Git](https://devcenter.heroku.com/articles/getting-started-with-java#store-your-app-in-git)

We now have the three major components of our app: build configuration and dependencies in pom.xml, process types in Procfile, and our application source in src/main/java/HelloWorld.java. Let’s put it into Git:

$ git init

$ git add .

$ git commit -m "init"

## [Deploy your application to Heroku](https://devcenter.heroku.com/articles/getting-started-with-java#deploy-your-application-to-heroku)

Create the app:

$ heroku create

Creating stark-sword-398... done, stack is cedar

http://stark-sword-398.herokuapp.com/ | git@heroku.com:stark-sword-398.git

Git remote heroku added

Deploy your code:

$ git push heroku master

Counting objects: 47, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (25/25), done.

Writing objects: 100% (47/47), 10.25 KiB, done.

Total 47 (delta 19), reused 42 (delta 17)

-----> Heroku receiving push

-----> Java app detected

-----> Installing OpenJDK 1.6... done

-----> Installing Maven 3.0.3... done

-----> Installing settings.xml... done

-----> executing /app/tmp/repo.git/.cache/.maven/bin/mvn -B -Duser.home=/tmp/build\_3k0p14ghrmdzs -Dmaven.repo.local=/app/tmp/repo.git/.cache/.m2/repository -s /app/tmp/repo.git/.cache/.m2/settings.xml -DskipTests=true clean install

[INFO] Scanning for projects...

[INFO]

[INFO] ------------------------------------------------------------------------

[INFO] Building helloworld 1.0-SNAPSHOT

[INFO] ------------------------------------------------------------------------

...

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------

[INFO] Total time: 10.062s

[INFO] Finished at: Tue Jan 31 23:27:20 UTC 2012

[INFO] Final Memory: 12M/490M

[INFO] ------------------------------------------------------------------------

-----> Discovering process types

Procfile declares types -> web

-----> Compiled slug size is 948K

-----> Launching... done, v3

http://empty-fire-6534.herokuapp.com deployed to Heroku

## [Visit your application](https://devcenter.heroku.com/articles/getting-started-with-java#visit-your-application)

You’ve deployed your code to Heroku, and specified the process types in a Procfile. You can now instruct Heroku to execute a process type. Heroku does this by running the associated command in a [dyno](https://devcenter.heroku.com/articles/dynos) - a lightweight container which is the basic unit of composition on Heroku.

Let’s ensure we have one dyno running the web process type:

$ heroku ps:scale web=1

You can check the state of the app’s dynos. The heroku ps command lists the running dynos of your application:

$ heroku ps

=== web: `java -cp target/classes:target/dep...`

web.1: up for 5s

Here, one dyno is running.

We can now visit the app in our browser with heroku open.

$ heroku open

Opening empty-fire-6534... done

## [Dyno sleeping and scaling](https://devcenter.heroku.com/articles/getting-started-with-java#dyno-sleeping-and-scaling)

Having only a single web dyno running will result in the dyno [going to sleep](https://devcenter.heroku.com/articles/dynos#dyno-sleeping) after one hour of inactivity. This causes a delay of a few seconds for the first request upon waking. Subsequent requests will perform normally.

To avoid this, you can scale to more than one web dyno. For example:

$ heroku ps:scale web=2

For each application, Heroku provides [750 free dyno-hours](https://devcenter.heroku.com/articles/usage-and-billing#750-free-dyno-hours-per-app). Running your app at 2 dynos would exceed this free, monthly allowance, so let’s scale back:

$ heroku ps:scale web=1

## [View the logs](https://devcenter.heroku.com/articles/getting-started-with-java#view-the-logs)

Heroku treats logs as streams of time-ordered events aggregated from the output streams of all the dynos running the components of your application. Heroku’s [Logplex](https://devcenter.heroku.com/articles/logplex) provides a single channel for all of these events.

View information about your running app using one of the [logging commands](https://devcenter.heroku.com/articles/logging), heroku logs:

$ heroku logs

2012-01-31T23:27:27+00:00 heroku[web.1]: Starting process with command `java -cp target/classes:target/dependency/\* HelloWorld`

2012-01-31T23:27:28+00:00 app[web.1]: 2012-01-31 23:27:28.280:INFO:oejs.Server:jetty-7.6.0.v20120127

2012-01-31T23:27:28+00:00 app[web.1]: 2012-01-31 23:27:28.334:INFO:oejsh.ContextHandler:started o.e.j.s.ServletContextHandler{/,null}

2012-01-31T23:27:28+00:00 app[web.1]: 2012-01-31 23:27:28.373:INFO:oejs.AbstractConnector:Started SelectChannelConnector@0.0.0.0:8236

2012-01-31T23:27:29+00:00 heroku[web.1]: State changed from starting to up

## [Next steps](https://devcenter.heroku.com/articles/getting-started-with-java#next-steps)

* The [Spring MVC Hibernate tutorial](https://devcenter.heroku.com/articles/spring-mvc-hibernate) will guide you through setting up a database-driven application on Heroku.
* Visit the [Java category](https://devcenter.heroku.com/categories/java) to learn more about developing and deploying Java applications.
* Read [How Heroku Works](https://devcenter.heroku.com/articles/how-heroku-works) for a technical overview of the concepts you’ll encounter while writing, configuring, deploying and running applications.

# How Heroku Works

*Last Updated: 12 August 2013*

### Table of Contents

* [Defining an application](https://devcenter.heroku.com/articles/how-heroku-works#defining-an-application)
* [Knowing what to execute](https://devcenter.heroku.com/articles/how-heroku-works#knowing-what-to-execute)
* [Deploying applications](https://devcenter.heroku.com/articles/how-heroku-works#deploying-applications)
* [Building applications](https://devcenter.heroku.com/articles/how-heroku-works#building-applications)
* [Running applications on dynos](https://devcenter.heroku.com/articles/how-heroku-works#running-applications-on-dynos)
* [Config vars](https://devcenter.heroku.com/articles/how-heroku-works#config-vars)
* [Releases](https://devcenter.heroku.com/articles/how-heroku-works#releases)
* [Dyno manager](https://devcenter.heroku.com/articles/how-heroku-works#dyno-manager)
* [Add-ons](https://devcenter.heroku.com/articles/how-heroku-works#add-ons)
* [Logging and monitoring](https://devcenter.heroku.com/articles/how-heroku-works#logging-and-monitoring)
* [HTTP routing](https://devcenter.heroku.com/articles/how-heroku-works#http-routing)
* [Tying it all together](https://devcenter.heroku.com/articles/how-heroku-works#tying-it-all-together)

This is a high-level, technical description of how Heroku works. It ties together many of the concepts you’ll encounter while writing, configuring, deploying and running applications on the Heroku platform.

Performing one of the [**Getting Started**](https://devcenter.heroku.com/articles/quickstart) tutorials will make the concepts in this documentation more concrete.

Read this document sequentially: in order to tell a coherent story, it incrementally unveils and refines the concepts describing the platform.

The final section ties all the definitions together, providing a [deploy-time](https://devcenter.heroku.com/articles/how-heroku-works#deploy)and [runtime-view](https://devcenter.heroku.com/articles/how-heroku-works#runtime) of Heroku.

## [Defining an application](https://devcenter.heroku.com/articles/how-heroku-works#defining-an-application)

Heroku lets you deploy, run and manage applications written in Ruby, Node.js, Java, Python, Clojure and Scala.

An application is a collection of source code written in one of these languages, perhaps a framework, and some dependency description that instructs a build system as to which additional dependencies are needed in order to build and run the application.

**Terminology** (Preliminary): Applications consist of your source code and a description of any dependencies.

Dependency mechanisms vary across languages: in Ruby you use aGemfile, in Python a requirements.txt, in Node.js a package.json, in Java a pom.xml and so on.

The source code for your application, together with the dependency file, should provide enough information for the Heroku platform to build your application, to produce something that can be executed.

## [Knowing what to execute](https://devcenter.heroku.com/articles/how-heroku-works#knowing-what-to-execute)

You don’t need to make many changes to an application in order to run it on Heroku. One requirement is informing the platform as to which parts of your application are runnable.

If you’re using some established framework, Heroku can figure it out. For example, in Ruby on Rails, it’s typically rails server, in Django it’spython <app>/manage.py runserver and in Node.js it’s the main field inpackage.json.

**Terminology**: [**Procfiles**](https://devcenter.heroku.com/articles/procfile) list process types - named commands that you may want executed.

For other applications, you may need to explicitly declare what can be executed. You do this in a text file that accompanies your source code - a [Procfile](https://devcenter.heroku.com/articles/procfile). Each line declares a [process type](https://devcenter.heroku.com/articles/process-model) - a named command that can be executed against your built application. For example, your Procfile may look like this:

web: java -jar lib/foobar.jar $PORT

queuty: java -jar lib/queue-processor.jar

This file declares a web process type and provides the command that needs to be executed in order to run it (in this case, java -jar lib/foobar.jar $PORT). It also declares a queuty process type, and its corresponding command.

The earlier definition of an application can now be refined to include this single additional Procfile.

**Terminology**: Applications consist of your source code, a description of any dependencies, and a Procfile.

Heroku is a polyglot platform – it lets you build, run and scale applications in a similar manner across all the languages – utilizing the dependencies and Procfile. The Procfile exposes an architectural aspect of your application (in the above example there are two entry points to the application) and this architecture lets you, for example, scale each part independently. An excellent guide to architecture principles that work well for applications running on Heroku can be found in [Architecting Applications for Heroku](https://devcenter.heroku.com/articles/architecting-apps).

## [Deploying applications](https://devcenter.heroku.com/articles/how-heroku-works#deploying-applications)

[Git](http://git-scm.com/) is a powerful, distributed version control system that many developers use to manage and version source code. The Heroku platform uses git as the primary means for deploying applications.

When you create an application on Heroku, it associates a new git remote, typically named heroku, with the local git repository for your application.

As a result, deploying code is just the familiar git push, but to theheroku remote instead:

$ git push heroku master

**Terminology**: Deploying applications involves sending the application to Heroku using git.

Deployment then, is about using git as a transport mechanism - moving your application from your local system to Heroku.

## [Building applications](https://devcenter.heroku.com/articles/how-heroku-works#building-applications)

When the Heroku platform receives a git push, it initiates a build of the source application. The build mechanism is typically language specific, but follows the same pattern, typically retrieving the specified dependencies, and creating any necessary assets (whether as simple as processing style sheets or as complex as compiling code).

**Advanced**: [**Buildpacks**](https://devcenter.heroku.com/articles/buildpacks) lie behind the slug compilation process. They’re open source - enabling you to extend Heroku to other languages and frameworks. Buildpacks take your application, its dependencies, and the language runtime, and produce slugs.

For example, when the build system receives a Rails application, it may fetch all the dependencies specified in the Gemfile, as well as generate files based on the asset pipeline. A Java application may fetch binary library dependencies using Maven, compile the source code together with those libraries, and produce a JAR file to execute.

The source code for your application, together with the fetched dependencies and output of the build phase such as generated assets or compiled code, as well as the language and framework, are assembled into a [slug](https://devcenter.heroku.com/articles/slug-compiler).

**Terminology**: A [**slug**](https://devcenter.heroku.com/articles/slug-compiler) is a bundle of your source, fetched dependencies, the language runtime, and compiled/generated output of the build system - ready for execution.

These slugs are a fundamental aspect of what happens during application execution - they contain your compiled, assembled application - ready to run - together with the instructions (the Procfile) of what you may want to execute.

## [Running applications on dynos](https://devcenter.heroku.com/articles/how-heroku-works#running-applications-on-dynos)

Heroku executes applications by running a command you specified in the Procfile, on a [dyno](https://devcenter.heroku.com/articles/dynos) that’s been preloaded with your prepared slug (in fact, with your release, which extends your slug and a few items not yet defined: config vars and add-ons).

Think of a running dyno as a lightweight, secure, virtualized Unix container that contains your application slug in its file system.

**Terminology**: [**Dynos**](https://devcenter.heroku.com/articles/dynos) are isolated, virtualized Unix containers, that provide the environment required to run an application.

Generally, if you deploy an application for the first time, Heroku will run 1 web dyno automatically. In other words, it will boot a dyno, load it with your slug, and execute the command you’ve associated with the web process type in your Procfile.

You have control over how many dynos are running at any given time. Given the Procfile example earlier, you can start 5 dynos, 3 for the web and 2 for the queuty process types, as follows:

$ heroku ps:scale web=3 queuty=2

When you deploy a new version of an application, all of the currently executing dynos are killed, and new ones (with the new release) are started to replace them - preserving the existing dyno formation.

**Terminology**: Your application’s [**dyno formation**](https://devcenter.heroku.com/articles/scaling#dyno-formation) is the total number of currently-executing dynos, divided between the various process types you have scaled.

To understand what’s executing, you just need to know what dynos are running which process types:

$ heroku ps

== web: 'java lib/foobar.jar $PORT'

web.1: up 2013/02/07 18:59:17 (~ 13m ago)

web.1: up 2013/02/07 18:52:08 (~ 20m ago)

web.2: up 2013/02/07 18:31:14 (~ 41m ago)

== queuty: `java lib/queue-processor.jar`

queuty.1: up 2013/02/07 18:40:48 (~ 32m ago)

queuty.2: up 2013/02/07 18:40:48 (~ 32m ago)

Dynos then, are an important means of scaling your application. In this example, the application is well architected to allow for the independent scaling of web and queue worker dynos.

## [Config vars](https://devcenter.heroku.com/articles/how-heroku-works#config-vars)

An application’s configuration is everything that is likely to vary between environments (staging, production, developer environments, etc.). This includes backing services such as databases, credentials, or environment variables that provide some specific information to your application.

Heroku lets you run your application with a customizable configuration - the configuration sits outside of your application code and can be changed independently of it.

The configuration for an application is stored in [config vars](https://devcenter.heroku.com/articles/config-vars). For example, here’s how to configure an encryption key for an application:

$ heroku config:add ENCRYPTION\_KEY= my\_secret\_launch\_codes

Adding config vars and restarting demoapp... done, v14

ENCRYPTION\_KEY: my\_secret\_launch\_codes

**Terminology**: [**Config vars**](https://devcenter.heroku.com/articles/config-vars) contain customizable configuration data that can be changed independently of your source code. The configuration is exposed to a running application via environment variables.

At runtime, all of the config vars are exposed as environment variables - so they can be easily extracted programatically. A Ruby application deployed with the above config var, can access it by callingENV["ENCRYPTION\_KEY"].

All dynos in an application will have access to the exact same set of config vars at runtime.

## [Releases](https://devcenter.heroku.com/articles/how-heroku-works#releases)

Earlier, this article stated that to run your application on a dyno, the Heroku platform loaded the dyno with your most recent slug. This needs to be refined: in fact it loads it with the slug and any config variables you have assigned to the application. The combination of slug and configuration is called a [release](https://devcenter.heroku.com/articles/releases).

**Terminology** (Preliminary): [**Releases**](https://devcenter.heroku.com/articles/releases) are an append-only ledger of slugs and config vars.

All releases are automatically persisted in an append-only ledger, making managing your application, and different releases, a cinch. Use the heroku releases command to see the audit trail of release deploys:

$ heroku releases

== demoapp Releases

v103 Deploy 582fc95 jon@heroku.com 2013/01/31 12:15:35

v102 Deploy 990d916 jon@heroku.com 2013/01/31 12:01:12

The number next to the deploy message, for example582fc95, corresponds to the commit hash of the repository you deployed to Heroku.

Every time you deploy a new version of an application, a new slug is created and release is generated.

As Heroku contains a store of the previous releases of your application, it’s very easy to rollback and deploy a previous release:

$ heroku releases:rollback v102

Rolling back demoapp... done, v102

$ heroku releases

== demoapp Releases

v104 Rollback to v102 jon@heroku.com 2013/01/31 14:11:33 (~15s ago)

v103 Deploy 582fc95 jon@heroku.com 2013/01/31 12:15:35

v102 Deploy 990d916 jon@heroku.com 2013/01/31 12:01:12

Making a material change to your application, whether it’s changing the source or configuration, results in a new release being created.

A release then, is the mechanism behind how Heroku lets you modify the configuration of your application (the config vars) independently of the application source (stored in the slug) - the release binds them together. Whenever you change a set of config vars associated with your application, a new release will be generated.

## [Dyno manager](https://devcenter.heroku.com/articles/how-heroku-works#dyno-manager)

Part of the Heroku platform, the [dyno manager](https://devcenter.heroku.com/articles/dynos), is responsible for keeping dynos running. For example, dynos are cycled at least once per day, or whenever the dyno manager detects a fault in the running application (such as out of memory exceptions) or problems with the underlying hardware that requires the dyno be moved to a new physical location.

**Terminology**: The [**dyno manager**](https://devcenter.heroku.com/articles/dynos) of the Heroku platform is responsible for managing dynos across all applications running on Heroku.

This dyno cycling happens transparently and automatically on a regular basis, and is logged.

**Terminology**: Applications with only a single web dyno [**sleep**](https://devcenter.heroku.com/articles/dynos#dyno-sleeping)after one hour of inactivity. When a sleeping application receives HTTP traffic, it will be awakened - causing a delay of a few seconds. Scaling the web dynos will avoid sleeping.

Because Heroku manages and runs applications, there’s no need to manage operating systems or other internal system configuration. [One-off dynos](https://devcenter.heroku.com/articles/one-off-dynos) can be run with their input/output attached to your local terminal. These can also be used to carry out admin tasks that modify the state of shared resources, for example database configuration - perhaps periodically through a [scheduler](https://devcenter.heroku.com/articles/scheduler).

**Terminology**: [**One-off Dynos**](https://devcenter.heroku.com/articles/one-off-dynos) are temporary dynos that can run with their input/output attached to your local terminal. They’re loaded with your latest release.

Here’s the simplest way to create and attach to a one-off dyno:

$ heroku run bash

Running `bash` attached to terminal... up, run.8963

~ $ ls

This will spin up a new dyno, loaded with your release, and then run thebash command - which will provide you with a unix shell (remember that dynos are effectively isolated virtualized unix containers). Once you’ve terminated your session, or after a period of inactivty, the dyno will be removed.

Changes to the filesystem on one dyno are not propagated to other dynos and are not persisted across deploys and dyno restarts. A better and more scalable approach is to use a shared resource such as a database or queue.

**Terminology**: Each dyno gets its own [**ephemeral filesystem**](https://devcenter.heroku.com/articles/dynos#ephemeral-filesystem)- with a fresh copy of the most recent release. It can be used as temporary scratchpad, but changes to the filesystem are not reflected to other dynos.

The ephemeral nature of the file system in a dyno can be demonstrated with the above command. If you create a one-off dyno by runningheroku run bash, the Unix shell on the dyno, and then create a file on that dyno, and then terminate your session - the change is lost. All dynos, even those in the same application, are isolated - and after the session is terminated the dyno will be killed. New dynos are always created from a slug, not from the state of other dynos.

## [Add-ons](https://devcenter.heroku.com/articles/how-heroku-works#add-ons)

Applications typically make use of add-ons to provide backing services such as databases, queueing & caching systems, storage, email services and more. Add-ons are provided as services by Heroku and third parties - there’s a large [marketplace](https://addons.heroku.com/) of add-ons you can choose from.

Heroku treats these add-ons as attached resources: provisioning an add-on is a matter of choosing one from the add-on marketplace, and attaching it to your application.

For example, here is how to add a Redis backing store add-on (by[RedisToGo](https://devcenter.heroku.com/articles/redistogo#provisioning-the-add-on)) to an application:

$ heroku addons:add redistogo:nano

The add-on service provider is responsible for the service - and the interface to your application is often provided through a config var. In this example, a REDISTOGO\_URL will be automatically added to your application when you provision the add-on. You can write code that connects to the service through the URL, for example:

uri = URI.parse(ENV["REDISTOGO\_URL"])

REDIS = Redis.new(:host => uri.host, :port => uri.port, :password => uri.password)

**Terminology**: [**Add-ons**](https://addons.heroku.com/) are third party, specialized, value-added cloud services that can be easily attached to an application, extending its functionality.

Add-ons are associated with an application, much like config vars - and so the earlier definition of a release needs to be refined. A release of your applications is not just your slug and config vars; it’s your slug, config vars as well as the set of provisioned add-ons.

**Terminology**: [**Releases**](https://devcenter.heroku.com/articles/releases) are an append-only ledger of slugs, config vars and add-ons. Heroku maintains an append-only ledger of releases you make.

Much like config vars, whenever you add, remove or change an add-on, a new release is created.

## [Logging and monitoring](https://devcenter.heroku.com/articles/how-heroku-works#logging-and-monitoring)

Heroku treats logs as streams of time-ordered events, and collates the stream of logs produced from all of the processes running in all dynos, and the Heroku platform components, into the [Logplex](https://devcenter.heroku.com/articles/logplex) - a high-performance, real-time system for log delivery.

It’s easy to examine the logs across all the platform components and dynos:

$ heroku logs

2013-02-11T15:19:10+00:00 heroku[router]: at=info method=GET path=/articles/custom-domains host=mydemoapp.heroku.com fwd=74.58.173.188 dyno=web.1 queue=0 wait=0ms connect=0ms service=1452ms status=200 bytes=5783

2013-02-11T15:19:10+00:00 app[web.2]: Started GET "/" for 1.169.38.175 at 2013-02-11 15:19:10 +0000

2013-02-11T15:19:10+00:00 app[web.1]: Started GET "/" for 2.161.132.15 at 2013-02-11 15:20:10 +0000

Here you see 3 timestamped log entries, the first from Heroku’s router, the last two from two dynos running the web process type.

**Terminology**: [**Logplex**](https://devcenter.heroku.com/articles/logplex) automatically collates log entries from all the running dynos of your app, as well as other components such as the routers, providing a single source of activity.

You can also dive into the logs from just a single dyno, and keep the channel open, listening for further events:

$ heroku logs --ps web.1 --tail

2013-02-11T15:19:10+00:00 app[web.2]: Started GET "/" for 1.169.38.175 at 2013-02-11 15:19:10 +0000

Logplex keeps a limited buffer of log entries solely for performance reasons. To persist them, and action events such as email notification on exception, use a [Logging Add-on](https://addons.heroku.com/#logging), which ties into log drains - an API for receiving the output from Logplex.

## [HTTP routing](https://devcenter.heroku.com/articles/how-heroku-works#http-routing)

Depending on your dyno formation, some of your dynos will be running the command associated with the web process type, and some will be running other commands associated with other process types.

The dynos that run process types named web are different in one way from all other dynos - they will receive HTTP traffic. Heroku’s [HTTP routers](https://devcenter.heroku.com/articles/http-routing) distributes incoming requests for your application across your running web dynos.

So scaling an app’s capacity to handle web traffic involves scaling the number of web dynos:

$ heroku ps:scale web+5

A random selection algorithm is used for HTTP request load balancing across web dynos - and this routing handles both HTTP and HTTPS traffic. It also supports multiple simultaneous connections, as well as timeout handling.

## [Tying it all together](https://devcenter.heroku.com/articles/how-heroku-works#tying-it-all-together)

The concepts explained here can be divided into two buckets: those that involve the development and deployment of an application, and those that involve the runtime operation of the Heroku platform and the application after its deployed.

The following two sections recapitulate the main components of the platform, separating them into these two buckets.

### [Deploy](https://devcenter.heroku.com/articles/how-heroku-works#deploy)

* Applications consist of your source code, a description of any dependencies, and a Procfile.
* [Procfiles](https://devcenter.heroku.com/articles/procfile) list process types - named commands that you may want executed.
* Deploying applications involves sending the application to Heroku using git.
* [Buildpacks](https://devcenter.heroku.com/articles/buildpacks) lie behind the slug compilation process. Buildpacks take your application, its dependencies, and the language runtime, and produce slugs.
* A [slug](https://devcenter.heroku.com/articles/slug-compiler) is a bundle of your source, fetched dependencies, the language runtime, and compiled/generated output of the build system - ready for execution.
* [Config vars](https://devcenter.heroku.com/articles/config-vars) contain customizable configuration data that can be changed independently of your source code. The configuration is exposed to a running application via environment variables.
* [Add-ons](https://addons.heroku.com/) are third party, specialized, value-added cloud services that can be easily attached to an application, extending its functionality.
* A [release](https://devcenter.heroku.com/articles/releases) is a combination of a slug (your application), config vars and add-ons. Heroku maintains an append-only ledger of releases you make.

### [Runtime](https://devcenter.heroku.com/articles/how-heroku-works#runtime)

* [Dynos](https://devcenter.heroku.com/articles/dynos) are isolated, virtualized unix containers, that provide the environment required to run an application.
* Your application’s [dyno formation](https://devcenter.heroku.com/articles/scaling#dyno-formation) is the total number of currently-executing dynos, divided between the various process types you have scaled.
* The [dyno manager](https://devcenter.heroku.com/articles/dynos) is responsible for managing dynos across all applications running on Heroku.
* Applications with only a single web dyno [sleep](https://devcenter.heroku.com/articles/dynos#dyno-sleeping) after one hour of inactivity by the dyno manager. Scaling to multiple web dynos will avoid this.
* [One-off Dynos](https://devcenter.heroku.com/articles/one-off-dynos) are temporary dynos that run with their input/output attached to your local terminal. They’re loaded with your latest release.
* Each dyno gets its own [ephemeral filesystem](https://devcenter.heroku.com/articles/dynos#ephemeral-filesystem) - with a fresh copy of the most recent release. It can be used as temporary scratchpad, but changes to the filesystem are not reflected to other dynos.
* [Logplex](https://devcenter.heroku.com/articles/logplex) automatically collates log entries from all the running dynos of your app, as well as other components such as the routers, providing a single source of activity.
* [Scaling](https://devcenter.heroku.com/articles/scaling) an application involves varying the number of dynos of each process type.

# [DEA 和 buildpack](http://limengyun.com/cloudfoundry/dea.html)

Thu 13 June 2013

By [limengyun](http://limengyun.com/author/limengyun.html)

In [backend](http://limengyun.com/category/backend.html).

tags: [cloudfoundry](http://limengyun.com/tag/cloudfoundry.html)

dea 和 buildpack是整个cloudfoundry 设计最出彩的地方，想想别的PaaS 开发新加一门语言是多么费劲？而cloudfoundry 仅仅需要几天，而且还支持用户自定义语言和应用类型， 这一切都是基于buildpack打包设计的功劳。

## 术语

在提到buildpack之前 有必要解释一下DEA，dea全称是 droplet execution agency，即执行droplet的代理。

droplet是cloudfoundry自创的一个概念，它是一个app的可运行实例配合实例启停脚本的压缩包。

举例来说，如果我把一个php应用放置在某个路径下，然后将apache配置好，最后写一个启动脚本，然后将apache, php应用代码和启动脚本打成一个压缩包。 在另外一台环境完全相同的机器上，你只需要下载这个压缩包，解压到对应目录下，然后启动脚本，应用就可以完美的复制到这台机器上。 这就是cloudfoundry进行动态扩容的原理和基础。 一个打好压缩包就是一个droplet，打压缩包的程序就叫buildpack，打包的过程叫staging。 不同的应用类型对应的buildpack代码也不同。如果要增加一门cloudfoundry默认不支持的语言或者应用类型，就需要自定义buildpack。

## dea职责

### staging和running droplet

在1.0中，staging 是由专门的组件stager来完成的，在2.0中去掉了stager改为直接在dea中进行staging。

这是dea最重要的两个职责。

官方专门针对staging和running的流程撰写了文档： [文章链接](http://docs.cloudfoundry.com/docs/running/architecture/how-applications-are-staged.html)

如何在众多dea中选择合适的dea来完成任务，是通过消息机制来实现的，参见在[NATS细节](http://limengyun.com/cloudfoundry/nats.html)中的例子“cloud controller指挥dea进行打包和运行”

### 向router注册使应用实例可以对外提供服务

dea上的应用流量无论有多大，对dea的影响都微乎其微。 因为dea不对外提供服务，dea控制的container才对外提供服务。 router和dea会定时通过NATS通信，dea将dea下的container的ip，host，port等消息报告给router，参见在[NATS细节](http://limengyun.com/cloudfoundry/nats.html)中的例子“向router注册”

### 查看应用文件

dea启动时会附带启动一个file api server，而dea directory server的启动则是单独进行的，代码在dea/go目录下。 这两个server的区别在于：

dea directory server 启动后会向router注册，即外部可以访问到dea directory server。 所有跟dea相关的上传（上传droplet）和下载（获取各种文件内容，如log文件）都是直接通过dea directory server来进行的。 file api server起一个验证并返回请求真实路径的作用

比如执行cf logs XXX -t命令，表面上看起来是客户端cf向cloud controller发起请求，但实际上是cloud controller 重定向到dea directory server来提供服务的

Getting logs **for** 23423 #0>>>

**REQUEST:** GET http:*//api.cf2.youdao.com/v2/apps/de48824b-8243-4218-8b52-d92c974453f8/instances/0/files/logs*

**REQUEST\_HEADERS:**

Authorization : bearer eyJhbGciOiJIUzI1NiJ9..cd11MxTrbCCpG\_5fU9\_DV1\_bE9Nz\_2lQ\_c1kari1WXI

Content-Length : 0

. RESPONSE: [302]

**RESPONSE\_HEADERS:**

connection : keep-alive

content-length : 0

content-type : application/json;charset=utf-8

date : Sun, 16 Jun 2013 08:51:37 GMT

location : http:*//882aa9de3cfa35c4e06a6f5613f0df2d.cf2.youdao.com/instance\_paths/b6d11bf2865189aafe0a4be4133688b7?hmac=cc48aa1aa305154ea4e49e8ee33c1ee4a3bbb9ddccd626b73f0a6ed9fe7d6d191a79ecf78b680e5e252ba8948e7ee444029e38ce216a9b828b3898e49dc49a06&path=logs&timestamp=1371372696*

server : nginx

x-frame-options : sameorigin

x-vcap-request-id : 76e95020-cd2c-4ba8-b41b-d99e8638e7aa

x-xss-protection : 1; mode=block

dea directory server 收到类似http://882aa9de3cfa35c4e06a6f5613f0df2d.cf2.youdao.com/instance\_paths/b6d11bf2865189aafe0a4be4133688b7?hmac=cc48aa1aa305154ea4e49e8ee33c1ee4a3bbb9ddccd626b73f0a6ed9fe7d6d191a79ecf78b680e5e252ba8948e7ee444029e38ce216a9b828b3898e49dc49a06&path=logs&timestamp=1371372696 这样的请求后会向file api server 发起请求，file api server根据hmac 判断请求是否合法，如果合法就返回log文件在操作系统上的真实地址，最后由dea directory server 返回log的内容。

## 配置

改一下nats的配置，然后新加一个配置项domain

**domain:** 你的平台域名。

这个配置项在默认配置中没有，但上面提到dea directory server会向router注册对外提供服务， 在lib/dea/directory\_server\_v2.rb提供给外部访问路径的时候如果domain值为nil，会导致directory server无法被访问。

def external\_hostname

"#{uuid}.#{domain}"

end

此外，还需要特别注意你的ruby安装路径，配置项中与ruby相关的路径要填对。

## 自定义buildpack支持

如果想要cloudfoundry支持一门新的语言或框架，自定义一个buildpack就可以了， 比如我们就将官方的java buildpack修改为支持ant+ivy编译，且从tomcat改为使用resin。 （J2EE架构下一个war包能在resin下跑不一定能在tomcat下跑） 这样就可以直接上传源代码而不用上传war包，且与公司的习惯保持一致。。

每一种语言只有一个buildpack，不同类别的应用打包在buildpack代码里面单独进行区分。 在buildpacks/vendor目录下，官方提供了三种语言（java，nodejs，ruby）的若干种应用类型的支持，同时也提供了一种方便添加自定义buildpack支持的机制。

新增添一个语言的buildpack需要实现以下三个脚本：

* ./bin/detect 检查同一种语言下应用的类型，例如是一个sinatra应用还是一个rails应用，以供compile脚本使用
* ./bin/compile 打包的主脚本，根据detect的结果选择打包的方式。
* ./bin/release 输出自定义的启动命令等相关信息，以便写入到最终的启动脚本。

dea会遍历所有的detect脚本，如果你编写的detect脚本满足条件，echo “应用类型的名字” exit 0 就可以了，cloudfoundry就会选择对应目录下的compile脚本执行。

以我们修改的java的compile脚本为例，compile的内容就是顺序执行

* 安装java
* 安装ant
* 安装resin
* 在应用目录想执行ant all 生成ROOT.war（应用build.xml必须支持ant all命令）
* cp ROOT.war 到 resin的webapps目录下
* 写一个脚本generate\_resin\_conf.rb，脚本的内容是根据传入的port参数动态的生成renin的resin.conf

最后我们需要完成release脚本，同样以修改的java为例，release文件的内容就是一个.yml文件，里面配置了一些启动命令和环境变量。 只需要吧启动命令修改为 ruby generate\_resin\_conf.rb && ./resin/bin/httpd.sh -c ../resin.conf

DONE

#### 注意1：

官方或开源第三方的buildpack在staging的时候基本都通过http下载一些东西，比如apache等等，关键在于这些地址都是在amazon s3上。 在天朝的网络下，采用这种方式就要看GFW的心情了，要正常使用最好把那些下载地址全部换成内网地址。

#### 注意2：

有些buildpack 会将/home/vcap/app 软连接到/app/app 然后再在/app/app中进行操作。 但之前在warden中提到过，在centos 6上使用默认文件系统实现的warden container 不具备完整的软连接功能（无法cd进入软连接后的目录，详情见warden的cent OS tip）， 所以想要在centos上正常使用仍然需要修改这部分代码。如果自己实现，就千万不要使用软连接。

#### 提示：

http://www.appfog.com 是一个基于cloudfoundry的paas，他自定义了许多语言的buildpack支持并在github上开源出来（github： https://github.com/appfog/ ）。