
Table of Contents

User Guide

Introduction	1.1
Getting Started	1.2
Running Apache Brooklyn	1.2.1
Deploying Blueprints	1.2.2
Monitoring and Managing Applications	1.2.3
Policies	1.2.4
Concepts	1.2.5
Downloads	1.3
Brooklyn Concepts	1.4
Entities	1.4.1
Application, Parent and Membership	1.4.2
Configuration, Sensors and Effectors	1.4.3
Lifecycle and ManagementContext	1.4.4
Dependent Configuration	1.4.5
Location	1.4.6
Policies	1.4.7
Execution	1.4.8
Stop/start/restart behaviour	1.4.9
Writing Blueprints	1.5
Creating YAML Blueprint	1.5.1
Entity Configuration	1.5.2
Setting Locations	1.5.3
Configuring VMs	1.5.4
Multiple Services and Dependency Injection	1.5.5
Custom Entities	1.5.6
Catalog	1.5.7
Catalog Items YAML Syntax	1.5.7.1
Templates and the Add-Application Wizard	1.5.7.2
Versioning	1.5.7.3
Catalog Management	1.5.7.4
Bundling	1.5.7.5
Brooklyn Server Command Line Arguments	1.5.7.6
Clusters, Specs, and Composition	1.5.8
Enrichers	1.5.9
Policies	1.5.10

Effectors	1.5.11
Clusters and Policies	1.5.12
Java Entities	1.5.13
Creating from a Maven Archetype	1.5.13.1
Defining and Deploying	1.5.13.2
Handling Bundle Dependencies	1.5.13.3
Topology, Dependencies, and Management Policies	1.5.13.4
Common Classes and Entities	1.5.13.5
Feeds	1.5.13.6
Writing an Entity	1.5.13.7
Custom Entity Development	1.5.13.8
Service State	1.5.13.9
Entitlements	1.5.13.10
Windows Blueprints	1.5.14
Winrm4j Client	1.5.14.1
Testing YAML Blueprints	1.5.15
Test Entities	1.5.15.1
Example Tests	1.5.15.2
Ansible in YAML Blueprints	1.5.16
About Ansible	1.5.16.1
Creating Blueprints with Ansible	1.5.16.2
Chef in YAML Blueprints	1.5.17
About Chef	1.5.17.1
Creating Blueprints from Chef	1.5.17.2
Writing Chef for Blueprints	1.5.17.3
Advanced Chef Integration	1.5.17.4
Salt in YAML Blueprints	1.5.18
About Salt	1.5.18.1
Creating Blueprints with Salt	1.5.18.2
YAML Blueprint Advanced Example	1.5.19
Blueprinting Tips	1.5.20
YAML Blueprint Reference	1.5.21
Deploying Blueprint	1.6
Clouds	1.6.1
Amazon Web Services	1.6.2
Azure Compute ARM	1.6.3
Azure Compute Classic	1.6.4
Apache CloudStack	1.6.5
Google Compute Engine	1.6.6
IBM Softlayer	1.6.7

OpenStack	1.6.8
Named Locations	1.6.9
Provisioned Machine Requirements	1.6.10
BYON	1.6.11
SSH Keys	1.6.12
Localhost	1.6.13
Location Customizers	1.6.14
Customizing Cloud Security Groups	1.6.15
Specialized Locations	1.6.16
Reference Guide	1.7
Production Installation	1.7.1
Starting, Stopping and Monitoring	1.7.2
Server CLI Reference	1.7.3
Client CLI Reference	1.7.4
CLI Reference Guide	1.7.4.1
CLI Usage Guide	1.7.4.2
GUI Guide	1.7.5
Launching	1.7.5.1
Deploying Blueprints	1.7.5.2
Monitoring and Managing Applications	1.7.5.3
Using Policies	1.7.5.4
REST API	1.7.6
Configuring Brooklyn	1.7.7
Memory Usage	1.7.7.1
Authentication	1.7.7.2
brooklyn.cfg	1.7.7.3
HTTPS Configuration	1.7.7.4
CORS Configuration	1.7.7.5
Persistence	1.7.8
High Availability	1.7.9
Configuring HA - an example	1.7.9.1
Logging	1.7.10
Externalized Configuration	1.7.11
Requirements	1.7.12
Upgrade	1.7.13
Security Guidelines	1.7.14
Troubleshooting	1.7.15
Overview	1.7.15.1
Web Console Issues	1.7.15.2
Deployment	1.7.15.3

Server Connectivity	1.7.15.4
Brooklyn Slow or Unresponsive	1.7.15.5
Increase Entropy	1.7.15.6
Increase System Resource Limits	1.7.15.7
Detailed Support Report	1.7.15.8
SoftwareProcess Entities	1.7.15.9
Going Deep in Java and Logs	1.7.15.10
Monitoring Memory Usage	1.7.15.11
Paths breakdown	1.7.16
Other 0.12.0 Resources	1.8
Glossary	1.8.1
Versions	1.8.2
Release Notes	1.8.3

Developer Guide

Get the Code	2.1
Maven Build	2.2
IDE Setup	2.3
Code Structure	2.4
Tests	2.5
License Considerations	2.6
Miscellaneous Tips and Tricks	2.7
Logging	2.8
Brooklyn Remote Debugging	2.9
GitHub	2.10
Javadoc	2.11
Other Resources	3.1

Apache Brooklyn 1.0.0-SNAPSHOT Documentation

Welcome to the Apache Brooklyn documentation.

Please select the section you wish to discover from the left menu. Alternatively, you can search within the entire documentation via the search field on the top left.

Getting Started

- [Running Apache Brooklyn](#)
- [Deploying Blueprints](#)
- [Monitoring and Managing Applications](#)
- [Policies](#)
- [Concepts](#)

Running Apache Brooklyn

This guide will walk you through deploying an example 3-tier web application to a public cloud, and demonstrate the autoscaling capabilities of the Brooklyn platform.

Two methods of deployment are detailed in this tutorial, using virtualisation with Vagrant and an install in your own environment (such as your local machine or in your private/public cloud).

The latter assumes that you have a [Java Runtime Environment \(JRE\)](#) installed (version 7 or later), as Brooklyn is Java under the covers.

To get you up-and-running quickly, the Vagrant option will provision four compute nodes for you to deploy applications to.

Install Apache Brooklyn

Installing with Vagrant

[Vagrant](#) is a software package which automates the process of setting up virtual machines (VM) such as [Oracle VirtualBox](#). We recommend it as the easiest way of getting started with Apache Brooklyn.

Firstly, download and install:

- [Vagrant](#)
- [Oracle VirtualBox](#)

Then download the provided Apache Brooklyn vagrant configuration from [here](#). This archive contains everything you need to create an environment for use with this guide, providing an Apache Brooklyn instance and some blank VMs.

Extract the `tar.gz` archive and navigate into the expanded `apache-brooklyn-1.0.0-SNAPSHOT-vagrant` folder (note: as this is a -SNAPSHOT version, your filename will be slightly different)

```
$ tar xvf apache-brooklyn-1.0.0-SNAPSHOT-vagrant.tar.gz
$ cd apache-brooklyn-1.0.0-SNAPSHOT-vagrant
```

Installing on CentOS & Red Hat Enterprise Linux 7

Please note, an RPM is not available for snapshot builds

For Centos 7 and RHEL 7 users, the recommended way to install Apache Brooklyn on RPM-based Linux distributions is by using the RPM package.

RPM is the de facto standard for packaging software on these Linux distributions and provides a mechanism for installing, upgrading and removing packages such as Apache Brooklyn. The RPM package contains all the necessary files associated with the Apache Brooklyn application.

This is a snapshot build and no RPM is available, please download [a different version](#).

Once downloaded, run the following shell command as root:

```
$ yum install apache-brooklyn-1.0.0-SNAPSHOT-1.rpm
```

Installing on Ubuntu 7 Debian

For Ubuntu and Debian users, the recommended way to install Apache Brooklyn is to use the deb file.

The deb file is the de facto standard for packaging software on these Linux distributions and provides a mechanism for installing, upgrading and removing packages such as Apache Brooklyn. The deb package contains all the necessary files associated with the Apache Brooklyn application.

Download the Apache Brooklyn [deb distribution](#).

Once downloaded, run the following shell command:

```
$ sudo dpkg -i apache-brooklyn_1.0.0-SNAPSHOT_noarch.deb
```

Installing on other Linux distributions, OSX and other UNIX-like platforms

For Linux or OSX please download the Apache Brooklyn `tar.gz` archive from the [download](#) section.

Extract the `tar.gz` archive (note: as this is a -SNAPSHOT version, your filename will be slightly different):

```
$ tar -zxf apache-brooklyn-dist-1.0.0-SNAPSHOT-timestamp-dist.tar.gz
$ cd apache-brooklyn-1.0.0-SNAPSHOT
```

Installing on Windows

For all versions of Microsoft Windows, please download the Apache Brooklyn zip file from [here](#).

Extract this zip file to a directory on your computer such as `c:\Program Files\brooklyn` where `c` is the letter of your operating system drive.

By default, no authentication is required and the web-console will listen on all network interfaces. For a production system, or if Apache Brooklyn is publicly reachable, it is strongly recommended to configure security. Documentation of configuration options include:

- [Security](#)
- [Persistence](#)
- [Cloud credentials](#)

Launch Apache Brooklyn

Important

Before launching Apache Brooklyn, please check the `date` on the local machine. Even several minutes before or after the actual time could cause problems.

Launching in Vagrant

Now start Apache Brooklyn with the following command:

```
$ vagrant up brooklyn
```


You can see if Apache Brooklyn launched OK by viewing the log files with the command

```
$ vagrant ssh brooklyn --command 'sudo journalctl -n15 -f -u brooklyn'
```

Launching on CentOS & Red Hat Enterprise Linux 7

Apache Brooklyn should now have been installed and be running as a system service. It can be stopped and started with the standard systemctl commands:

```
$ systemctl start|stop|restart|status brooklyn
```

The application should then output its logs to `brooklyn.debug.log` and `brooklyn.info.log`, please refer to the [paths](#) page for the locations of these.

Launching on Ubuntu & Debian

Apache Brooklyn should now have been installed and be running as a system service. It can be stopped and started with the standard service commands:

```
$ sudo service brooklyn start|stop|restart|status
```

The application should then output its logs to `brooklyn.debug.log` and `brooklyn.info.log`, please refer to the [paths](#) page for the locations of these.

Launching on other Linux distributions, OSX and other UNIX-like platforms

Now start Apache Brooklyn with the following command:

```
$ bin/start
```

The application should then output its log to `brooklyn.debug.log` and `brooklyn.info.log`, please refer to the [paths](#) page for the locations of these.

Launching on Windows

You can now start Apache Brooklyn by running `c:\Program Files\brooklyn\bin\start.bat`

The application should then output its log into the console and also `c:\Program Files\brooklyn\data\log\brooklyn.debug.log` and `c:\Program Files\brooklyn\data\log\brooklyn.info.log`

Control Apache Brooklyn

Apache Brooklyn has a web console which can be used to control the application. The Brooklyn log will contain the address of the management interface:

```
INFO Started Brooklyn console at http://127.0.0.1:8081/, running
classpath://brooklyn.war
```

By default it can be accessed by opening [127.0.0.1:8081](#) in your web browser.

The rest of this getting started guide uses the Apache Brooklyn command line interface (CLI) tool, `br`. This tool is both distributed with Apache Brooklyn or can be downloaded from [here](#).

For details on the CLI, see the [Client CLI Reference](#) page.

Deploying Blueprints

Blueprints are descriptors or patterns which describe how Apache Brooklyn should deploy applications. Blueprints are written in [YAML](#) and many of the entities available are defined in the [Brooklyn Catalog](#).

Launching from a Blueprint

We'll start by deploying an application with a simple [YAML blueprint](#) containing an [Apache Tomcat](#) server.

Copy the [blueprint](#) below into a text file, "myapp.yaml", in your workspace (Note, to copy the file you can hover your mouse over the right side of the text box below to get a Javascript "copy" button).

```
name: Tomcat
services:
- type: org.apache.brooklyn.entity.webapp.tomcat.TomcatServer
  name: tomcatServer
  location: <your-location-definition-goes-here>
```

Locations

Before you can create an application with this configuration, you need to modify the [YAML](#) to specify a [location](#). Locations in Apache Brooklyn are server resources which Brooklyn can use to deploy applications. These locations may be servers or cloud providers which provide access to servers.

In order to configure the [location](#) in which Apache Brooklyn launches an application, replace the `location:` element with values for your chosen target environment. Here are some examples of the various [location](#) types:

Vagrant location

The Vagrant configuration described in [Running Apache Brooklyn](#), on the previous page is the recommended way of running this tutorial. This configuration comes with four blank vagrant configurations called byon1 to byon4.

These can be launched by entering the following command into the terminal in the vagrant configuration directory.

```
$ vagrant up byon1 byon2 byon3 byon4
```

The [location](#) in "myapp.yaml" can now be replaced with the following [YAML](#) to launch using these vagrant servers.

```
location:
  byon:
    user: vagrant
    password: vagrant
    hosts:
      - 10.10.10.101
      - 10.10.10.102
      - 10.10.10.103
      - 10.10.10.104
```

A cloud location

Apache Brooklyn uses [Apcahe jclouds](#) to support a range of cloud locations. More information on the range of providers and configurations is available [here](#).

As an example, here is a configuration for [Amazon Web Services \(AWS\)](#). Swap the identity and credential with your AWS account details, then replace the [location](#) in your "myapp.yaml" with this.

```
location:
  jclouds:aws-ec2:
    identity: ABCDEFGHIJKLMNOPQRST
    credential: s3cr3tsq1rr3ls3cr3tsq1rr3ls3cr3tsq1rr3l
```

"Bring your own nodes" location

The Bring Your Own Nodes (BYON) configuration allows Apache Brooklyn to make use of already available servers. These can be specified by a list of IP addresses with a user and password as shown below. More information including the full range of configuration options is available [here](#).

Replace the hosts, user and password in the example below with your own server details, then replace the [location](#) in your "myapp.yaml" with this.

```
location:
  byon:
    user: myuser
    password: mypassword
    # or...
    #privateKeyFile: ~/.ssh/my.pem
  hosts:
    - 192.168.0.18
    - 192.168.0.19
```

Note: For instructions on setting up a variety of locations or storing credentials/locations in a file on disk rather than in the [blueprint](#), see [Locations](#) in the Operations section of the User Guide.

Deploying the Application

First, log in to brooklyn with the command line interface (CLI) tool by typing:

```
$ br login http://localhost:8081/
```

To secure the Apache Brooklyn instance, you can add a username and password to Brooklyn's properties file, as described in the User Guide [here](#). If this is configured, the login command will require an additional parameter for the userid and will then prompt for a password.

Now you can create the application with the command below:

```
$ br deploy myapp.yaml
Id:      hTPAF19s
Name:    Tomcat
Status:  In progress
```

Depending on your choice of [location](#) it may take some time for the application to start, the next page describes how you can monitor the progress of the application deployment and verify if it was successful.

Monitoring and Managing Applications

So far we have gone through Apache Brooklyn's ability to *deploy* an application [blueprint](#) to a [location](#), but this just the beginning. Next we will outline how to *manage* the application that has been deployed.

Applications

Having created the application, we can find a summary of all deployed applications using:

```
$ br application
Id      Name      Status  Location
hTPAF19s Tomcat    RUNNING ajVVAhER
```

`application` can be shortened to the alias `app`, for example:

```
$ br app
Id      Name      Status  Location
hTPAF19s Tomcat    RUNNING ajVVAhER
```

A full list of abbreviations such as this can be found in the [CLI reference guide](#).

In the above example the Id `hTPAF19s` and the Name `Tomcat` are shown. You can use either of these handles to monitor and control the application. The Id shown for your application will be different to this but the name should be the same, note that if you are running multiple applications the Name may not be unique.

Things we might want to do

Get the application details

Using the name `Tomcat` we can get the application details:

```
$ br application Tomcat
Id:          hTPAF19s
Name:        Tomcat
Status:      RUNNING
ServiceUp:   true
Type:        org.apache.brooklyn.entity.stock.BasicApplication
CatalogItemId: null
LocationId:  ajVVAhER
LocationName: FixedListMachineProvisioningLocation:ajVV
LocationSpec: vagrantbyon
LocationType: org.apache.brooklyn.location.byon.FixedListMachineProvisioningLocation
```

Explore the hierarchy of all applications

We can explore the management hierarchy of all applications, which will show us the entities they are composed of.

```
$ br tree
|- Tomcat
+- org.apache.brooklyn.entity.stock.BasicApplication
  |- tomcatServer
  +- org.apache.brooklyn.entity.webapp.tomcat.TomcatServer
```

View our application's blueprint

You can view the [blueprint](#) for the application again:

```
$ br application Tomcat spec
name: Tomcat
location: mylocation
services:
- type: brooklyn.entity.webapp.tomcat.TomcatServer
```

View our application's configuration

You can view the configuration of the application:

```
$ br application Tomcat config
Key                               Value
camp.template.id                 l67i25CM
brooklyn.wrapper_app             true
```

Entities

An [Entity](#) is Apache Brooklyn's representation of a software package or service which it can control or interact with. All of the entities Apache Brooklyn can use are listed in the [Brooklyn Catalog](#).

To list the entities of the application you can use the `entity` or `ent` command:

```
$ br application Tomcat entity
Id          Name          Type
Wx7r1C4e    tomcatServer    org.apache.brooklyn.entity.webapp.tomcat.TomcatServer
```

This shows one [entity](#) is available: `tomcatServer`. Note that this is the name we gave the [entity](#) in the [YAML](#) in [Launching from a Blueprint](#) on the previous page.

You can get summary information for this [entity](#) by providing its name (or ID).

```
$ br application Tomcat entity tomcatServer
Id:          Wx7r1C4e
Name:        tomcatServer
Status:      RUNNING
ServiceUp:   true
Type:        org.apache.brooklyn.entity.webapp.tomcat.TomcatServer
CatalogItemId: null
```

Also you can see the configuration of this [entity](#) with the `config` command.

```
$ br application Tomcat entity tomcatServer config
Key                               Value
jmx.agent.mode                   JMXMP_AND_RMI
brooklyn.wrapper_app             true
camp.template.id                 yBcQuFZe
onbox.base.dir                   /home/vagrant/brooklyn-managed-processes
onbox.base.dir.resolved          true
install.unique_label             TomcatServer_7.0.65
```

Sensors

Sensors are properties which show the state of an *entity* and provide a real-time picture of an *entity* within an application.

You can view the sensors available on the application using:

```
$ br application Tomcat sensor
Name          Description
Value
service.isUp  Whether the service is active and availability (confirmed and monitored)
true
service.notUp.indicators A map of namespaced indicators that the service is not up
{}
service.problems A map of namespaced indicators of problems with a service
{}
service.state   Actual lifecycle state of the service
"RUNNING"
service.state.expected Last controlled change to service state, indicating what the expected state should be
"running @ 1450356994928 / Thu Dec 17 12:56:34 GMT 2015"
```

To explore sensors on a specific *entity* use the `sensor` command with an *entity* specified:

```
$ br application Tomcat entity tomcatServer sensor
Name          Description
Value
download.addon.urls URL patterns for downloading named add-ons (will substitute things like ${version} automatically)
download.url       URL pattern for downloading the installer (will substitute things like ${version} automatically)
"http://download.nextag.com/apache/tomcat/tomcat-7/v${version}/bin/apache-tomcat-${version}.tar.gz"
expandedinstall.dir Directory for installed artifacts (e.g. expanded dir after unpacking .tgz)
"/home/vagrant/brooklyn-managed-processes/installs/TomcatServer_7.0.65/apache-tomcat-7.0.65"
host.address      Host IP address
"10.10.10.101"
host.name         Host name
"10.10.10.101"
host.sshAddress   user@host:port for ssh'ing (or null if inappropriate)
"vagrant@10.10.10.101:22"
host.subnet.address Host address as known internally in the subnet where it is running (if different to host.name)
"10.10.10.101"
host.subnet.hostname Host name as known internally in the subnet where it is running (if different to host.name)
"10.10.10.101"
# etc. etc.
```

To display the value of a selected *sensor*, give the command the *sensor* name as an argument

```
$ br application Tomcat entity tomcatServer sensor webapp.url
"http://10.10.10.101:8080/"
```

Effectors

Effectors are a means by which you can manipulate the entities in an application. You can list the available effectors for your application using:

```
$ br application Tomcat effector
Name          Description          Parameters
restart       Restart the process/service represented by an entity
start         Start the process/service represented by an entity locations
stop          Stop the process/service represented by an entity
```


For example, to stop an application, use the `stop` **effector**. This will cleanly shutdown all components in the application and return any cloud machines that were being used. Note that the three "lifecycle" related effectors, `start`, `stop`, and `restart`, are common to all applications and software process entities in Brooklyn.

You can list the effectors for a specific **entity** using the command:

```
$ br application Tomcat entity tomcatServer effector
Name          Parameters          Description
deploy        url,targetName       Deploys the given artifact, from a source URL, to a given deployment filename
/context
populateServiceNotUpDiagnostics indicators          Populates the attribute service.notUp.diagnostics, with any available health
restart        restartChildren,restartMachine Restart the process/service represented by an entity
start          locations            Start the process/service represented by an entity
stop           stopProcessMode,stopMachineMode Stop the process/service represented by an entity
undeploy       targetName            Undeploys the given context/artifact
```

To view the details for a specific **effector**, append it's name to the command:

```
$ br application Tomcat entity tomcatServer effector deploy
Name      Description          Parameters
deploy    Deploys the given artifact, from a source URL, to a given deployment filename/context url,targetName
```

These effectors can also be invoked by appending `invoke` to this command. Some effectors require parameters for their invocation. For example, if we look at the details for `deploy` above we can see it requires a `url` and `targetName`.

These parameters can be supplied using `--param parm=value` or just `-P parm=value`.

The commands below deploy the Apache Tomcat **hello world example** to our Tomcat Server. In these commands, a variable is created for the root URL using the appropriate **sensor** and the index page html is displayed.

```
$ br application Tomcat entity tomcatServer effector deploy invoke -P url=https://tomcat.apache.org/tomcat-6.0-
doc/appdev/sample/sample.war -P targetName=sample
$ webapp=$(br application Tomcat entity tomcatServer sensor webapp.url | tr -d '"')
$ curl $webapp/sample/
<html>
<head>
<title>Sample "Hello, World" Application</title>
</head>
...
```

Note that at present a `tr` command is required in the second line below to strip quotation characters from the returned **sensor** value.

Activities

Activities are the actions an application or **entity** takes within Apache Brooklyn. The `activity` command allows us to list out these activities.

To view a list of all activities associated with an **entity** enter:

```
$ br application Tomcat entity tomcatServer activity
Id          Task          Submitted          Status          Streams
```

LtD5P1cb	start	Thu Dec 17 15:04:43 GMT 2015	Completed	
l2qo4vTl	provisioning (FixedListMachineProvisi...	Thu Dec 17 15:04:43 GMT 2015	Completed	
wLD764HE	pre-start	Thu Dec 17 15:04:43 GMT 2015	Completed	
KLTXDkoa	ssh: initializing on-box base dir ./b...	Thu Dec 17 15:04:43 GMT 2015	Completed	env,stderr,std
in,stdout				
jwwcJWmF	start (processes)	Thu Dec 17 15:04:43 GMT 2015	Completed	
...				

To view the details of an individual activity, add its ID to the command. In our case this is `jwwcJWmF`

```
$ br application Tomcat entity tomcatServer activity jwwcJWmF
Id: jwwcJWmF
DisplayName: start (processes)
Description:
EntityId: efUvVWAw
EntityDisplayName: TomcatServer:efUv
Submitted: Thu Dec 17 15:04:43 GMT 2015
Started: Thu Dec 17 15:04:43 GMT 2015
Ended: Thu Dec 17 15:08:59 GMT 2015
CurrentStatus: Completed
IsError: false
IsCancelled: false
SubmittedByTask: LtD5P1cb
Streams:
DetailedStatus: "Completed after 4m 16s

No return value (null)"
```

Things we might want to do

View Input and Output Streams

If an activity has associated input and output streams, these may be viewed by providing the activity scope and using the commands, "env", "stdin", "stdout", and "stderr". For example, for the "initializing on-box base dir" activity from the result of the earlier example,

```
$ br application Tomcat entity tomcatServer act KLTXDkoa stdout
BASE_DIR_RESULT:/home/vagrant/brooklyn-managed-processes:BASE_DIR_RESULT
```

Monitor the progress of an effector

To monitor progress on an application as it deploys, for example, one could use a shell loop:

```
$ while br application Tomcat entity tomcatServer activity | grep 'In progress' ; do
  sleep 1; echo ; date;
done
```

This loop will exit when the application has deployed successfully or has failed. If it fails then the 'stderr' command may provide information about what happened in any activities that have associated streams:

```
$ br application Tomcat entity tomcatServer act KLTXDkoa stderr
```

Diagnose a failure

If an activity has failed, the "DetailedStatus" value will help us diagnose what went wrong by showing information about the failure.

```
$ br application evHULq0n entity tomcatServer activity 1ZZ9x662
Id:                1ZZ9x662
DisplayName:       post-start
Description:
EntityId:         qZeyoITy
EntityDisplayName: tomcatServer
Submitted:        Mon Jan 25 12:54:55 GMT 2016
Started:         Mon Jan 25 12:54:55 GMT 2016
Ended:          Mon Jan 25 12:59:56 GMT 2016
CurrentStatus:    Failed
IsError:         true
IsCancelled:     false
SubmittedByTask:  hwU7Qvgm
Streams:
DetailedStatus:   "Failed after 5m: Software process entity TomcatServerImpl{id=qZeyoITy} did not pass is-running check within the required 5m limit (5m elapsed)

java.lang.IllegalStateException: Software process entity TomcatServerImpl{id=qZeyoITy} did not pass is-running check within the required 5m limit (5m elapsed)
    at org.apache.brooklyn.entity.software.base.SoftwareProcessImpl.waitForEntityStart(SoftwareProcessImpl.java:586)
    at org.apache.brooklyn.entity.software.base.SoftwareProcessImpl.postDriverStart(SoftwareProcessImpl.java:260)
    at org.apache.brooklyn.entity.software.base.SoftwareProcessDriverLifecycleEffectorTasks.postStartCustom(SoftwareProcessDriverLifecycleEffectorTasks.java:169)
    at org.apache.brooklyn.entity.software.base.lifecycle.MachineLifecycleEffectorTasks$PostStartTask.run(MachineLifecycleEffectorTasks.java:570)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
    at org.apache.brooklyn.util.core.task.DynamicSequentialTask$DstJob.call(DynamicSequentialTask.java:342)
    at org.apache.brooklyn.util.core.task.BasicExecutionManager$SubmissionCallable.call(BasicExecutionManager.java:468)
    at java.util.concurrent.FutureTask.run(FutureTask.java:266)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)"
```

Adding the "--children" or "-c" parameter will show the activity's child activities, to allow the hierarchical structure of the activities to be investigated:

```
$ br application Tomcat entity tomcatServer activity -c jwwcJWmF
```

Id	Task	Submitted	Status
UpYRc3fw	copy-pre-install-resources	Thu Dec 17 15:04:43 GMT 2015	Completed
ig8sBHQr	pre-install	Thu Dec 17 15:04:43 GMT 2015	Completed
Elp4HaVj	pre-install-command	Thu Dec 17 15:04:43 GMT 2015	Completed
Y0vNobJk	setup	Thu Dec 17 15:04:43 GMT 2015	Completed
VN3cDKki	copy-install-resources	Thu Dec 17 15:08:43 GMT 2015	Completed
xDJXQC0j	install	Thu Dec 17 15:08:43 GMT 2015	Completed
zxMDXUxz	post-install-command	Thu Dec 17 15:08:58 GMT 2015	Completed
qnQnw70c	customize	Thu Dec 17 15:08:58 GMT 2015	Completed
ug044ArS	copy-runtime-resources	Thu Dec 17 15:08:58 GMT 2015	Completed
STavcRc8	pre-launch-command	Thu Dec 17 15:08:58 GMT 2015	Completed
HKrYfH6h	launch	Thu Dec 17 15:08:58 GMT 2015	Completed
T1m8Vxbq	post-launch-command	Thu Dec 17 15:08:59 GMT 2015	Completed
n8eK5USE	post-launch	Thu Dec 17 15:08:59 GMT 2015	Completed

Scopes in CLI commands

Many commands require a "scope" expression to indicate the target on which they operate. The scope expressions are as follows (values in brackets are aliases for the scope):

- `application` APP-ID (app, a)
Selects an application, e.g. "br application myapp"
- `entity` ENT-ID (ent, e)
Selects an `entity` within an application scope, e.g. br application myapp entity myserver
- `effector` EFF-ID (eff, f)
Selects an `effector` of an `entity` or application, e.g. br a myapp e myserver eff xyz
- `config` CONF-KEY (conf, con, c)
Selects a configuration key of an `entity` e.g. br a myapp e myserver config jmx.agent.mode
- `activity` ACT-ID (act, v)
Selects an activity of an `entity` e.g. br a myapp e myserver act iHG7sq1

For example

```
$ br application Tomcat entity tomcatServer config
```

runs the `config` command with application scope of `Tomcat` and `entity` scope of `tomcatServer` .

Policies

A Clustered Example

We'll now look at a more complex example that better shows the capabilities of Brooklyn, including management of a running clustered application.

Below is the annotated [blueprint](#). Download the blueprint into a text file, `mycluster.yaml`, in your workspace. Before you create an application with it, review and/or change the the [location](#) where the application will be deployed.

You will need four machines for this example: one for the load-balancer (nginx), and three for the Tomcat cluster (but you can reduce this by changing the `maxPoolSize` below).

```
name: Tomcat Cluster

location:
  byon:
    user: vagrant
    password: vagrant
    hosts:
      - 10.10.10.101
      - 10.10.10.102
      - 10.10.10.103
      - 10.10.10.104

services:
- type: org.apache.brooklyn.entity.group.DynamicCluster
  name: Cluster
  id: cluster
  brooklyn.config:
    cluster.initial.size: 1
    dynamiccluster.memberspec:
      $brooklyn:entitySpec:
        type: org.apache.brooklyn.entity.webapp.tomcat.TomcatServer
        name: Tomcat Server
        brooklyn.config:
          wars.root: http://search.maven.org/remotecontent?filepath=org/apache/brooklyn/example/brooklyn-exampl
e-hello-world-webapp/0.8.0-incubating/brooklyn-example-hello-world-webapp-0.8.0-incubating.war

        brooklyn.policies:
          - type: org.apache.brooklyn.policy.ha.ServiceRestarter
            brooklyn.config:
              failOnRecurringFailuresInThisDuration: 5m
          brooklyn.enrichers:
            - type: org.apache.brooklyn.policy.ha.ServiceFailureDetector
              brooklyn.config:
                entityFailed.stabilizationDelay: 30s

        brooklyn.policies:
          - type: org.apache.brooklyn.policy.ha.ServiceReplacer

          - type: org.apache.brooklyn.policy.autoscaling.AutoScalerPolicy
            brooklyn.config:
              metric: webapp.reqs.perSec.perNode
              metricUpperBound: 3
              metricLowerBound: 1
              resizeUpStabilizationDelay: 2s
              resizeDownStabilizationDelay: 1m
              maxPoolSize: 3

        brooklyn.enrichers:
```

```

- type: org.apache.brooklyn.enricher.stock.Aggregator
  brooklyn.config:
    enricher.sourceSensor: $brooklyn:sensor("webapp.reqs.perSec.windowed")
    enricher.targetSensor: $brooklyn:sensor("webapp.reqs.perSec.perNode")
    enricher.aggregating.fromMembers: true
    transformation: average

- type: org.apache.brooklyn.entity.proxy.nginx.NginxController
  name: Load Balancer (nginx)
  brooklyn.config:
    loadbalancer.serverpool: $brooklyn:entity("cluster")
    nginx.sticky: false

```

The Tomcat Cluster

The `dynamicCluster` can dynamically increase or decrease the number of members. Resizing the cluster can either be carried out manually via effectors or automatically by attaching an `AutoScalerPolicy`.

It is configured with a `dynamiccluster.memberspec`, which defines the type and configuration of members in the cluster. In our example, each is a Tomcat server with a WAR deployed at the root URL.

Deploy the app:

```

$ br deploy mycluster.yaml
Id:      nGY58ZZN
Name:    Tomcat Cluster
Status:  In progress

```

And wait for the app to be running, viewing its state with:

```

$ br application
Id      Name      Status  Location
nGY58ZZN Tomcat Cluster RUNNING  Mf0CJac6

```

You can view the list of entities within the cluster with the command below (which drills into the application named "Tomcat Cluster", then into its child `entity` named "Cluster", and then lists its entities):

```

$ br application "Tomcat Cluster" entity "Cluster" entity
Id      Name      Type
dYfUvLIw quarantine org.apache.brooklyn.entity.group.QuarantineGroup
tOpMeYYr Tomcat Server org.apache.brooklyn.entity.webapp.tomcat.TomcatServer

```

The "quarantine" `entity` is used when Tomcat servers fail to start correctly - this `entity` is by default added to the quarantine group, where it can later be investigated. This can be disabled using the configuration `dynamiccluster.quarantineFailedEntities: false`.

Tomcat auto-repair

Each Tomcat `entity` has a `ServiceFailureDetector` `enricher` and a `ServiceRestarter` `policy`.

An `enricher` generates new events or `sensor` values (metrics) for the `entity` by modifying or aggregating data from one or more other sensors. A `policy` coordinates the runtime management of entities by carrying out actions, initiated by specific triggers. Policies are often used to keep the system in a healthy state, such as handling failures and auto-scaling.

The built-in functionality of the Tomcat `entity` is to use the "service.state" `sensor` to report its status. It will be "on-fire" when a failure is detected, or "running" when healthy, or one of the other lifecycle states such as "starting", "stopping" or "stopped".

The `ServiceFailureDetector` `enricher` emits an "Entity Failed" event whenever a failure is detected, and similarly an "Entity Recovered" event when recovered. The configuration option `serviceOnFireStabilizationDelay` will suppress reporting of failure until the `entity` is detected as failed for the given duration. This is very useful so as not to over-react to temporary failures.

The `ServiceRestarter` `policy` attaches to an `entity` in order to restart the service on failure. If there are subsequent failures within a configurable time interval, or if the restart fails, the service `entity` is marked as failed and no further restarts are attempted.

Try killing the Tomcat process for one of the members in the cluster. The command below will kill Tomcat on the vagrant VMs named "byon1" to "byon4":

```
for i in byon{1..4}; do
  vagrant ssh ${i} --command 'ps aux | grep -i tomcat | grep -v grep | awk '{print $2}' | xargs kill -9'
done
```

You can view the state of the Tomcat server with the command below (which drills into the application named "Tomcat Cluster", then into its child `entity` named "Cluster", and then into the first member of the cluster named "Tomcat Server"):

```
$ br application "Tomcat Cluster" entity "Cluster" entity "Tomcat Server"
Id:                tOpMeYYr
Name:              Tomcat Server
Status:            ON_FIRE
ServiceUp:         false
Type:              org.apache.brooklyn.entity.webapp.tomcat.TomcatServer
CatalogItemId:     org.apache.brooklyn.entity.webapp.tomcat.TomcatServer:0.0.0.SNAPSHOT
```

After the given `stabilizationDelay`, the Tomcat server will be automatically restarted.

Cluster auto-replace

The cluster has a `ServiceReplacer` `policy`. This attaches to a `DynamicCluster` and replaces a failed member; if this fails, it sets the Cluster state to on-fire.

To simulate a terminal failure of a member, repeatedly kill the process (using the command above).

The Tomcat server should be replaced by a new member of the cluster, and then the old failed member removed.

You can view the list of Tomcat servers in the cluster with the command below (which drills into the application named "Tomcat Cluster", then into its child `entity` named "Cluster", and then lists the child entities):

```
$ br application "Tomcat Cluster" entity "Cluster" entity
Id      Name      Type
dYfUvLIw quarantine org.apache.brooklyn.entity.group.QuarantineGroup
tOpMeYYr Tomcat Server org.apache.brooklyn.entity.webapp.tomcat.TomcatServer
mgoRpkKH Tomcat Server org.apache.brooklyn.entity.webapp.tomcat.TomcatServer
```

Auto-scaling

The `AutoScalerPolicy` attaches to a `Resizable` entity (in this case the `dynamiccluster`) and dynamically adjusts its size in response to keeping a metric within a given range. It adds/removes members (e.g. Tomcat instances) automatically.

In this example policy, the `metricUpperBound` for requests per second per server is configured very low at just 3. This makes it simple to trigger auto-scaling. The `resizeUpStabilizationDelay` of 2 seconds means the load must be sustained for at least that length of time. The

`resizeDownStabilizationDelay` of 1 minute means there must be low load for a full one minute before it will scale back down.

Find the URL of the load-balancer - see the `NginxController` sensor named `main.uri`.

To generate load, you can use your web-browser by repeatedly refreshing that page. Alternatively, you could use a load generator like jmeter, or use a script such as the one shown below (changing URL for the URL of your load-balancer):

```
URL=http://10.10.10.101:8000/
for i in {1..600}; do
  for j in {1..50}; do
    curl --fail --silent ${URL} > /dev/null || echo "Curl failed with exit code $?"
  done
  echo "Finished batch $i"
  sleep 1
done
```

While those curl commands run in a separate terminal, you can look at the metrics for the first Tomcat server using the command:

```
$ br application "Tomcat Cluster" entity "Cluster" entity "Tomcat Server" sensor
Name                                     Value                                     Description
...
webapp.reqs.perSec.last                  0.9980039920159681                      Reqs/sec (last datapoint)
webapp.reqs.perSec.windowed              0.9326571333555038                      Reqs/sec (over time window)
webapp.reqs.processingTime.fraction.last entage, last datapoint) "0.067%"         Fraction of time spent processing, reported by webserver (perc
webapp.reqs.processingTime.fraction.windowed entage, over time window) "0.073%"         Fraction of time spent processing, reported by webserver (perc
webapp.reqs.processingTime.max          ver (millis) ""                          Max processing time for any single request, reported by webser
webapp.reqs.processingTime.total        "3.12s"                                Total processing time, reported by webserver (millis)
webapp.reqs.total                        5575                                    Request count
webapp.tomcat.connectorStatus           "STARTED"                              Catalina connector state name
webapp.url                              "http://10.10.10.103:18082/"           URL
...
```

You can look at the average requests per second on the cluster with the command:

```
$ br application "Tomcat Cluster" entity "Cluster" sensor "webapp.reqs.perSec.perNode"
25.765557404326124
```

When this value exceeds 3 for two seconds, the cluster will scale up. You can see the new instance using the command:

```
$ br application "Tomcat Cluster" entity "Cluster" entity
Id      Name      Type
dYfUvLIw quarantine org.apache.brooklyn.entity.group.QuarantineGroup
mgoRpKH Tomcat Server org.apache.brooklyn.entity.webapp.tomcat.TomcatServer
```


xpLeJufy	Tomcat Server	org.apache.brooklyn.entity.webapp.tomcat.TomcatServer
CpabLxZE	Tomcat Server	org.apache.brooklyn.entity.webapp.tomcat.TomcatServer

Cancel the curl commands (or wait for them to finish), and then wait for the one minute `resizeDownStabilizationDelay` .
The cluster will scale back to the minimum one instance.

```
$ br application "Tomcat Cluster" entity "Cluster" entity
Id      Name      Type
dYfUvLIw quarantine org.apache.brooklyn.entity.group.QuarantineGroup
mgoRpKKH Tomcat Server org.apache.brooklyn.entity.webapp.tomcat.TomcatServer
```

Concepts

The following section provides a quick summary of the main Brooklyn concepts you will encounter in Getting Started. For further discussion of these concepts see [The Theory Behind Brooklyn](#), and the detailed descriptions in [Brooklyn Concepts](#).

Deployment and Management Brooklyn is built for agile deployment of applications across cloud and other targets, and real-time [autonomic](#) management. "[Autonomic](#) computing" is the concept of components looking after themselves where possible (self-healing, self-optimizing, etc).

Blueprints A [blueprint](#) defines an application by specifying its components, such as processes, or combinations of processes across multiple machines and services. The [blueprint](#) also specifies the inter-relationships between the configurations of the components.

Entities The central concept in a Brooklyn deployment is that of an [entity](#). An [entity](#) represents a resource under management (individual machines or software processes) or logical collections of these. Entities are arranged hierarchically. They can have events, operations, and processing logic associated with them, and it is through this mechanism that the active management is delivered.

Applications are the top level entities that are the parents of all other entities.

Configuration Entities can have arbitrary configuration values, which get inherited by their child entities. You can set global (Brooklyn-wide) properties in (`~/.brooklyn/brooklyn.properties`). Common configuration keys have convenient aliases called "flags".

Sensors are the mechanism for entities to expose information for other entities to see. Sensors from an [entity](#) can be subscribed to by other entities to track changes in the [entity](#)'s activity. Sensors can be updated, potentially frequently, by the [entity](#) or associated tasks.

Effectors are the mechanism for entities to expose the operations that can be invoked on it by other entities. The invoker is able to track the execution of that [effector](#) with tasks.

Lifecycle The management context of Brooklyn associates a "lifecycle" with Brooklyn entities. Common operations are start, stop, and restart (whose meaning differs slightly for applications and processes; the details are in the concepts guide linked above). Starting an application results in the `start()` operation being performed recursively (typically in parallel) on the application's children.

Tasks Lifecycle and other operations in Brooklyn are tracked as tasks. This allows current and past processing to be observed by operators, and processing to be managed across multiple management nodes.

Locations can be defined in order to specify where the processes of an application will run. Brooklyn supports different cloud providers and pre-prepared machines (including localhost), known as "BYON" (Bring Your Own Nodes).

Policies Policies perform the active management enabled by Brooklyn. Entities can have [Policy](#) instances attached to them, which can subscribe to sensors from other entities or run periodically. When they run they can perform calculations, look up other values, invoke effectors or emit [sensor](#) values from the [entity](#) with which they are associated.

Enrichers These are mechanisms that subscribe to a [sensor](#), or multiple sensors, and output a new [sensor](#). For example, the [enricher](#) which sums a [sensor](#) across multiple entities (used to get the total requests-per-second for all the web servers in a cluster), and the [enricher](#) which calculates a 60-second rolling average.

Downloads -

The downloads on this page have not been voted on and should be used at your own risk. The latest stable release can be accessed on the [main download page](#).

Download Version 1.0.0-SNAPSHOT

<https://dist.apache.org/repos/dist/release/brooklyn/apache-brooklyn-0.12.0/apache-brooklyn-0.12.0>

Download	File/Format	checksums (?)
Binary distribution Server & client	apache-brooklyn-1.0.0-SNAPSHOT-bin.tar.gz	SHA1
	apache-brooklyn-1.0.0-SNAPSHOT-bin.zip	SHA1
RPM package CentOS7, RHEL7, etc.	apache-brooklyn-1.0.0-SNAPSHOT-1.noarch.rpm	SHA1
DEB package Ubuntu, Debian, etc.	apache-brooklyn-1.0.0-SNAPSHOT.deb	SHA1
Client CLI only	apache-brooklyn-1.0.0-SNAPSHOT-client-cli-linux.tar.gz	SHA1
	apache-brooklyn-1.0.0-SNAPSHOT-client-cli-linux.zip	SHA1
	apache-brooklyn-1.0.0-SNAPSHOT-client-cli-macosx.tar.gz	SHA1
	apache-brooklyn-1.0.0-SNAPSHOT-client-cli-macosx.zip	SHA1
	apache-brooklyn-1.0.0-SNAPSHOT-client-cli-windows.tar.gz	SHA1
	apache-brooklyn-1.0.0-SNAPSHOT-client-cli-windows.zip	SHA1
Source code	apache-brooklyn-1.0.0-SNAPSHOT-src.tar.gz	SHA1
	apache-brooklyn-1.0.0-SNAPSHOT-src.zip	SHA1

Release Notes

Release notes can be found [here](#).

Maven

If you use Maven, you can add Brooklyn with the following in your pom:

```
<!-- include all Brooklyn items in our project -->
<dependencies>
  <dependency>
    <groupId>org.apache.brooklyn</groupId>
    <artifactId>brooklyn-all</artifactId>
    <version>1.0.0-SNAPSHOT</version>
  </dependency>
</dependencies>
```

`brooklyn-all` brings in all dependencies, including jclouds. If you prefer a smaller repo you might want just `brooklyn-core` , `brooklyn-policies` , and some of `brooklyn-software-webapp` , `brooklyn-software-database` , `brooklyn-software-messaging` , or others (browse the full list [here](#)).

If you wish to use the Apache snapshot repo, you can add this to your `pom.xml` :

```
<!-- include repos for snapshot items and other dependencies -->
<repositories>
  <repository>
    <id>apache-nexus-snapshots</id>
    <name>Apache Nexus Snapshots</name>
    <url>https://repository.apache.org/content/repositories/snapshots</url>
    <releases> <enabled>false</enabled> </releases>
    <snapshots> <enabled>true</enabled> </snapshots>
  </repository>
</repositories>
```

Please note: to add a snapshot version of Brooklyn as a dependency to your project, you must either have Brooklyn built locally or one of these snapshot repositories in your POM.

Source Code

Source code is hosted at github.com/apache/brooklyn, with this version in branch `master`. These locations have a `README.md` in the root which explains how to get the code including submodules.

Useful information on working with the source is [here](#).

Brooklyn Concepts

This introduces brooklyn and describes how it simplifies the deployment and management of big applications. It is intended for people who are using brooklyn-supported application components (such as web/app servers, data stores) to be able to use brooklyn to easily start their application in multiple locations with off-the-shelf management policies.

- [Entities](#)
- [Application, Parent and Membership](#)
- [Configuration, Sensors and Effectors](#)
- [Lifecycle and ManagementContext](#)
- [Dependent Configuration](#)
- [Location](#)
- [Policies](#)
- [Execution](#)
- [Stop/start/restart behaviour](#)

Entities

The central concept in a Brooklyn deployment is that of an **entity**. An **entity** represents a resource under management, either *base* entities (individual machines or software processes) or logical collections of these entities.

Fundamental to the processing model is the capability of entities to be the *parent* of other entities (the mechanism by which collections are formed), with every **entity** having a single parent **entity**, up to the privileged top-level **application entity**.

Entities are code, so they can be extended, overridden, and modified. Entities can have events, operations, and processing logic associated with them, and it is through this mechanism that the active management is delivered.

The main responsibilities of an **entity** are:

- Provisioning the **entity** in the given **location** or locations
- Holding configuration and state (attributes) for the **entity**
- Reporting monitoring data (sensors) about the status of the **entity**
- Exposing operations (effectors) that can be performed on the **entity**
- Hosting management policies and tasks related to the **entity**

Application, Parent and Membership

All entities have a **parent** *entity*, which creates and manages it, with one important exception: *applications*. Application entities are the top-level entities created and managed externally, manually or programmatically.

Applications are typically defined in Brooklyn as an **application descriptor**. This is a Java class specifying the entities which make up the application, by extending the class `AbstractApplication`, and specifying how these entities should be configured and managed.

All entities, including applications, can be the parent of other entities. This means that the "child" is typically started, configured, and managed by the parent. For example, an application may be the parent of a web cluster; that cluster in turn is the parent of web server processes. In the management console, this is represented hierarchically in a tree view.

A parallel concept is that of **membership**: in addition to one fixed parent, and *entity* may be a **member** of any number of special entities called **groups**. Membership of a group can be used for whatever purpose is required; for example, it can be used to manage a collection of entities together for one purpose (e.g. wide-area load-balancing between locations) even though they may have been created by different parents (e.g. a multi-tier stack within a *location*).

Configuration, Sensors and Effectors

Configuration

All entities contain a map of config information. This can contain arbitrary values, typically keyed under static `configKey` fields on the `Entity` sub-class. These values are inherited, so setting a configuration value at the application level will make it available in all entities underneath unless it is overridden.

Configuration is propagated when an application "goes live" (i.e. it becomes "managed", either explicitly or when its `start()` method is invoked), so config values must be set before this occurs.

Configuration values can be specified in a configuration file (`brooklyn.cfg`) to apply universally, and/or programmatically to a specific `entity` and its descendants by calling `.configure(KEY, VALUE)` in the `entity` spec when creating it. There is also an `entity.config().set(KEY, VALUE)` method.

Additionally, many common configuration parameters are available as "flags" which can be supplied as Strings when constructing then `entity`, in the form `EntitySpec.create(MyEntity.class).configure("config1", "value1").configure("config2", "value2")`.

Documentation of the flags available for individual entities can normally be found in the javadocs. The `@SetFromFlag` annotations on `configKey` static field definitions in the `entity`'s interface is the recommended mechanism for exposing configuration options.

Sensors and Effectors

Sensors (activity information and notifications) and **effectors** (operations that can be invoked on the `entity`) are defined by entities as static fields on the `Entity` subclass.

Sensors can be updated by the `entity` or associated tasks, and sensors from an `entity` can be subscribed to by its parent or other entities to track changes in an `entity`'s activity.

Effectors can be invoked by an `entity`'s parent remotely, and the invoker is able to track the execution of that `effector`. Effectors can be invoked by other entities, but use this functionality with care to prevent too many managers!

An `entity` consists of a Java interface (used when interacting with the `entity`) and a Java class. For resilience, it is recommended to store the `entity`'s state in attributes (see `getAttribute(AttributeKey)`). If internal fields can be used then the data will be lost on brooklyn restart, and may cause problems if the `entity` is to be moved to a different brooklyn management node.

Lifecycle and ManagementContext

Under-the-covers, at heart of the brooklyn management plane is the `ManagementContext`. This is started automatically when using launching an application using the brooklyn CLI. For programmatic use, see

```
BrooklynLauncher.newLauncher().launch().
```

A Brooklyn deployment consists of many entities in a hierarchical tree, with the privileged *application* `entity` at the top level.

An application `entity` (`Application` class) is responsible for starting the deployment of all its child entities (i.e. the entire `entity` tree under its ownership).

An `Application`'s `start()` method begins provisioning the child entities of the application (and their entities, recursively).

Provisioning of entities typically happens in parallel automatically, although this can be customized. This is implemented as **tasks** which are tracked by the management plane and is accessible in the web-based management console and REST API.

Customized provisioning can be useful where two starting entities depend on each other. For example, it is often necessary to delay start of one `entity` until another `entity` reaches a certain state, and to supply run-time information about the latter to the former.

When new entities are created, the `entity` is wired up to an application by giving it a parent. The `entity` is then explicitly "managed", which allows other entities to discover it.

Typically a Brooklyn deployment has a single management context which records:

- all entities under management that are reachable by the application(s) via the parent-child relationships,
- the state associated with each `entity`,
- subscribers (listeners) to `sensor` events arising from the entities,
- active tasks (jobs) associated with any the `entity`,
- which Brooklyn management node is mastering (managing) each `entity`.

In a multi-`location` deployment, management operates in all regions, with brooklyn `entity` instances being mastered in the relevant region.

When management is distributed a Brooklyn deployment may consist of multiple Brooklyn management nodes each with a `ManagementContext` instance.

Dependent Configuration

Under the covers Brooklyn has a sophisticated [sensor](#) event and subscription model, but conveniences around this model make it very simple to express cross-[entity](#) dependencies. Consider the example where Tomcat instances need to know the URL of a database (or a set of URLs to connect to a Monterey processing fabric, or other entities)

```
setConfiguration(UsesJava.JAVA_OPTIONS, ImmutableMap.of("mysql.url",
    attributeWhenReady(mysql, MySqlConnection.MY_SQL_URL) ))
```

The `attributeWhenReady(Entity, Sensor)` call (a static method on the class `DependentConfiguration`) causes the configuration value to be set when that given [entity](#)'s attribute is ready. In the example, `attributeWhenReady()` causes the JVM system property `mysql.url` to be set to the value of the `MySqlConnection.MY_SQL_URL` [sensor](#) from `mysql` when that value is ready. As soon as the database URL is announced by the `MySqlConnection` [entity](#), the configuration value will be available to the Tomcat cluster.

By default "ready" means being *set* (non-null) and, if appropriate, *non-empty* (for collections and strings) or *non-zero* (for numbers). Formally the interpretation of ready is that of "Groovy truth" defined by an `asBoolean()` method on the class and in the Groovy language extensions.

You can customize "readiness" by supplying a `Predicate` (Google common) or `closure` (Groovy) in a third parameter. This evaluates candidate values reported by the [sensor](#) until one is found to be `true`. For example, passing `{ it.size()>=3 }` as the readiness argument would require at least three management plane URLs.

More information on this can be found in the javadoc for `DependentConfiguration`, along with a few other methods such as `valueWhenAttributeReady` which allow post-processing of an attribute value.

Note that if the value of `CONFIG_KEY` passed to `Entity.getConfig` is a Closure or Task (such as returned by `attributeWhenReady`), the first access of `Entity.getConfig(CONFIG_KEY)` will block until the task completes. Typically this does the right thing, blocking when necessary to generate the right start-up sequence without the developer having to think through the order, but it can take some getting used to. Be careful not to request config information until really necessary (or to use non-blocking "raw" mechanisms), and in complicated situations be ready to attend to circular dependencies. The management console gives useful information for understanding what is happening and resolving the cycle.

Location

Entities can be provisioned/started in the **location** of your choice. Brooklyn transparently uses **jclouds** to support different cloud providers and to support BYON (Bring Your Own Nodes).

The implementation of an **entity** (e.g. Tomcat) is agnostic about where it will be installed/started. When writing the application definition specify the **location** or list of possible locations (`Location` instances) for hosting the **entity**.

`Location` instances represent where they run and indicate how that **location** (resource or service) can be accessed.

For example, a `JBoss7Server` will usually be running in an `SshMachineLocation` , which contains the credentials and address for sshing to the machine. A cluster of such servers may be running in a `MachineProvisioningLocation` , capable of creating new `SshMachineLocation` instances as required.

Policies

Policies perform the active management enabled by Brooklyn. Entities can have zero or more `Policy` instances attached to them.

Policies can subscribe to sensors from entities or run periodically, and when they run they can perform calculations, look up other values, and if deemed necessary invoke effectors or emit `sensor` values from the `entity` with which they are associated.

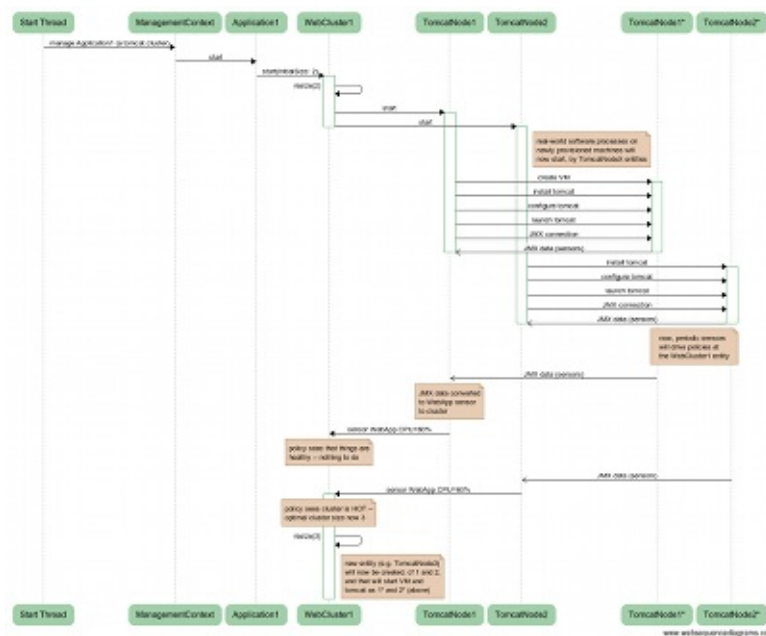
Execution

All processing, whether an **effector** invocation or a **policy** cycle, are tracked as **tasks**. This allows several important capabilities:

- active and historic processing can be observed by operators
- the invocation context is available in the thread, to check entitlement (permissions) and maintain a hierarchical causal chain even when operations are run in parallel
- processing can be managed across multiple management nodes

Some executions create new entities, which can then have tasks associated with them, and the system will record, for example, that a start **effector** on the new **entity** is a task associated with that **entity**, with that task created by a task associated with a different **entity**.

The execution of a typical overall start-up sequence is shown below:



Integration

One vital aspect of Brooklyn is its ability to communicate with the systems it starts. This is abstracted using a **driver** facility in Brooklyn, where a driver describes how a process or service can be installed and managed using a particular technology.

For example, a `TomcatServer` may implement start and other effectors using a `TomcatSshDriver` which inherits from `JavaSoftwareProcessSshDriver` (for JVM and JMX start configuration), inheriting from `AbstractSoftwareProcessSshDriver` (for SSH scripting support).

Particularly for sensors, some technologies are used so frequently that they are packaged as **feeds** which can discover their configuration (including from drivers). These include JMX and HTTP (see `JmxFeed` and `HttpFeed`).

Brooklyn comes with **entity** implementations for a growing number of commonly used systems, including various web application servers, databases and NoSQL data stores, and messaging systems.

Stop/start/restart behaviour

Many entities expose `start`, `stop` and `restart` effectors. The semantics of these operations (and the parameters they take) depends on the type of `entity`.

Top-level applications

A top-level application is a grouping of other entities, pulling them together into the "application" of your choice. This could range from a single app-server, to an app that is a composite of a no-sql cluster (e.g. MongoDB sharded cluster, or Cassandra spread over multiple datacenters), a cluster of load-balanced app-servers, message brokers, etc.

start(Collection <Location>)

This will start the application in the given `location(s)`. Each child-`entity` within the application will be started concurrently, passing the `location(s)` to each child. The start `effector` will be called automatically when the application is deployed through the catalog. It is strongly recommended to not call start again.

stop()

Stop will terminate the application and all its child entities (including releasing all their resources). The application will also be unmanaged, **removing** it from Brooklyn.

restart()

This will invoke `restart()` on each child-`entity` concurrently (with the default values for the child-`entity`'s restart `effector` parameters). It is strongly recommended to not call this, unless the application has been explicitly written to implement restart.

Software Process (e.g MySql, Tomcat, JBoss app-server, MongoDB)

start(Collection <Location>)

This will start the software process in the given `location`. If a machine `location` is passed in, then the software process is started there. If a cloud `location` is passed in, then a new VM will be created in that cloud - the software process will be **installed+launched** on that new VM.

The start `effector` will have been called automatically when deploying an application through the catalog. In normal usage, do not invoke start again.

If calling `start()` a second time, with no locations passed in (e.g. an empty list), then it will go through the start sequence on the existing `location` from the previous call. It will **install+customize+launch** the process. For some entities, this could be *dangerous*. The customize step might execute a database initialisation script, which could cause data to be overwritten (depending how the initialisation script was written).

If calling `start()` a second time with additional locations, then these additional locations will be added to the set of locations. In normal usage it is not recommended. This could be desired behaviour if the `entity` had previously been entirely stopped (including its VM terminated) - but for a simple one-`entity` app then you might as well have deleted the entire app and created a new one.

stop(boolean stopMachine)

If `stopMachine==true`, this `effector` will stop the software process and then terminate the VM (if a VM had been created as part of `start()`). This behaviour is the inverse of the first `start()` `effector` call. When stopping the software process, it does not uninstall the software packages / files.

If `stopMachine==false`, this `effector` will stop just the software process (leaving the VM and all configuration files / install artifacts in place).

restart(boolean restartMachine, boolean restartChildren)

This will restart the software process.

If `restartMachine==true`, it will also terminate the VM and create a new VM. It will then install+customize+launch the software process on the new VM. It is equivalent of invoking `stop(true)` and then `start(Collections.EMPTY_LIST)`. If `restartMachine==false`, it will first attempt to stop the software process (which should be a no-op if the process is not running), and will then start the software process (without going through the **install+customize** steps).

If `restartChildren==true`, then after restarting itself it will call `restart(restartMachine, restartChildren)` on each child-`entity` concurrently.

Recommended operations

The recommended operations to invoke to stop just the software process, and then to restart it are:

- Select the software process `entity` in the tree (*not* the parent application, but the child of that application).
- Invoke `stop(stopMachine=false)`
- Invoke `restart(restartMachine=false, restartChildren=false)`

Writing Blueprints

- [Creating YAML Blueprint](#)
- [Entity Configuration](#)
- [Setting Locations](#)
- [Configuring VMs](#)
- [Multiple Services and Dependency Injection](#)
- [Custom Entities](#)
- [Catalog](#)
- [Clusters, Specs, and Composition](#)
- [Enrichers](#)
- [Policies](#)
- [Effectors](#)
- [Clusters and Policies](#)
- [Java Entities](#)
- [Windows Blueprints](#)
- [Testing YAML Blueprints](#)
- [Ansible in YAML Blueprints](#)
- [Chef in YAML Blueprints](#)
- [Salt in YAML Blueprints](#)
- [YAML Blueprint Advanced Example](#)
- [Blueprinting Tips](#)
- [YAML Blueprint Reference](#)

Creating YAML Blueprint

A First Blueprint

The easiest way to write a [blueprint](#) is as a [YAML](#) file. This follows the [OASIS CAMP](#) plan specification, with some extensions described below. (A [YAML reference](#) has more information, and if the [YAML](#) doesn't yet do what you want, it's easy to add new extensions using your favorite JVM language.)

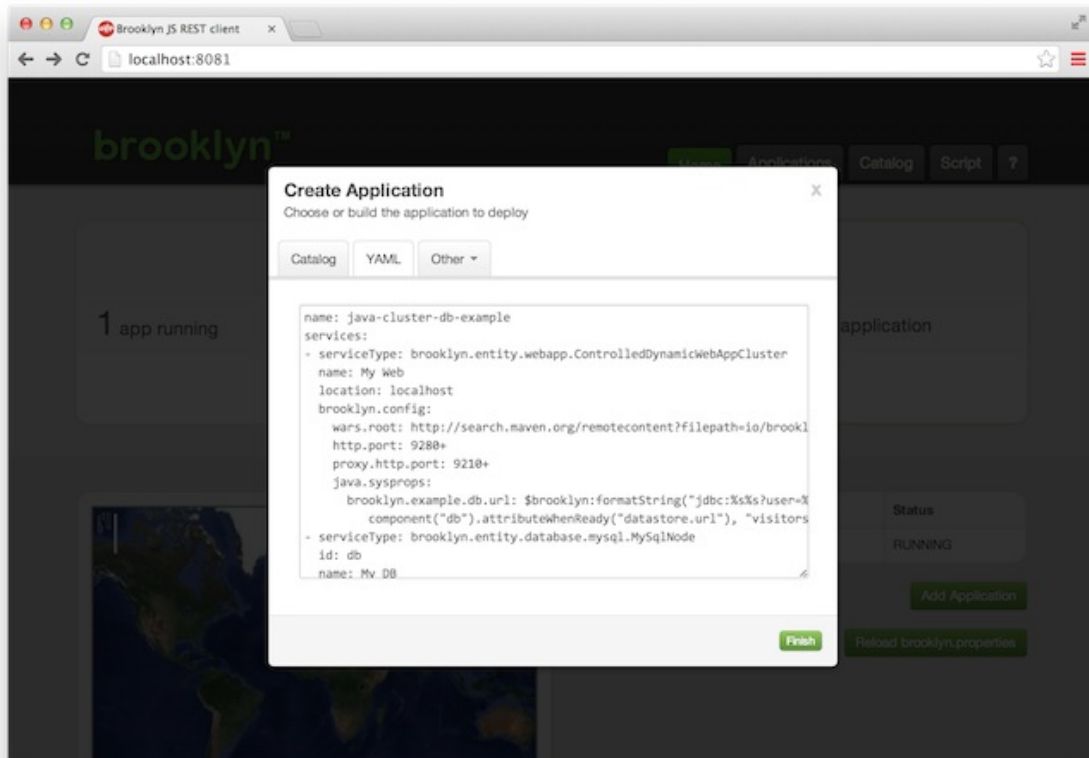
The Basic Structure

Here's a very simple [YAML blueprint](#) plan, to explain the structure:

```
name: simple-appserver
location: localhost
services:
- type: org.apache.brooklyn.entity.webapp.jboss.JBoss7Server
```

- The `name` is just for the benefit of us humans.
- The `location` specifies where this should be deployed. If you've [set up passwordless localhost SSH access](#) you can use `localhost` as above, but if not, just wait ten seconds for the next example.
- The `services` block takes a list of the typed services we want to deploy. This is the meat of the [blueprint](#) plan, as you'll see below.

Finally, the clipboard in the top-right corner of the example plan box above (hover your cursor over the box) lets you easily copy-and-paste into the web-console: simply [download and launch](#) Brooklyn, then in the "Create Application" dialog at the web console (usually <http://127.0.0.1:8081/>, paste the copied [YAML](#) into the "Yaml" tab of the dialog and press "Finish". There are several other ways to deploy, including `curl` and via the command-line, and you can configure users, https, persistence, and more, as described [in the ops guide](#).



More Information

Topics to explore next on the topic of [YAML](#) blueprints are:

Plenty of examples of blueprints exist in the Brooklyn codebase, so another starting point is to `git clone` it and search for `*.yaml` files therein.

Brooklyn lived as a Java framework for many years before we felt confident to make a declarative front-end, so you can do pretty much anything you want to by dropping to the JVM. For more information on Java:

- start with a [Maven archetype](#)
- see all [Brooklyn Java guide](#) topics
- look at test cases in the [codebase](#)

You can also come talk to us, on IRC (#brooklyncentral on Freenode) or any of the usual [hailing frequencies](#), as these documents are a work in progress.

Entity Configuration

Within a [blueprint](#) or catalog item, entities can be configured. The rules for setting this configuration, including when composing and extending existing entities, is described in this section.

Basic Configuration

Within a [YAML](#) file, [entity](#) configuration should be supplied within a `brooklyn.config` map. It is also possible to supply configuration at the top-level of the [entity](#). However, that approach is discouraged as it can sometimes be ambiguous (e.g. if the config key is called "name" or "type"), and also it does not work in all contexts such as for an [enricher](#)'s configuration.

A simple example is shown below:

```
services:
- type: org.apache.brooklyn.entity.webapp.tomcat.TomcatServer
  brooklyn.config:
    webapp.enabledProtocols: http
    http.port: 9080
    wars.root: http://search.maven.org/remotecontent?filepath=org/apache/brooklyn/example/brooklyn-example-hello-world-webapp/0.9.0/brooklyn-example-hello-world-webapp-0.9.0.war
```

If no config value is supplied, the default for that config key will be used. For example, `http.port` would default to 8080 if not explicitly supplied.

Some config keys also have a short-form (e.g. `httpPort` instead of `http.port` would also work in the [YAML](#) example above). However, that approach is discouraged as it does not work in all contexts such as for inheriting configuration from a parent [entity](#).

Configuration in a Catalog Item

When defining an [entity](#) in the catalog, it can include configuration values like any other [blueprint](#) (i.e. inside the `brooklyn.config` block).

It can also explicitly declare config keys, using the `brooklyn.parameters` block. The example below illustrates the principle:

```
brooklyn.catalog:
  items:
  - id: entity-config-example
    itemType: entity
    name: Entity Config Example
    item:
      type: org.apache.brooklyn.entity.software.base.VanillaSoftwareProcess
      brooklyn.parameters:
      - name: custom.message
        type: string
        description: Message to be displayed
        default: Hello
      brooklyn.config:
        shell.env:
          MESSAGE: $brooklyn:config("custom.message")
        launch.command: |
          echo "My example launch command: $MESSAGE"
        checkRunning.command: |
          echo "My example checkRunning command: $MESSAGE"
```

Once added to the catalog, it can be used with the simple [blueprint](#) below (substituting the [location](#) of your choice). Because no configuration has been overridden, this will use the default value for `custom.message`, and will use the given values for `launch.command` and `checkRunning.command`:

```
location: aws-ec2:us-east-1
services:
- type: entity-config-example
```

For details of how to write and add catalog items, see [Catalog](#).

Config Key Constraints

The config keys in the `brooklyn.parameters` can also have constraints defined, for what values are valid. If more than one constraint is defined, then they must all be satisfied. The constraints can be any of:

- `required`: deployment will fail if no value is supplied for this config key.
- `regex: ...`: the value will be compared against the given regular expression.
- A predicate, declared using the DSL `$brooklyn:object`.

This is illustrated in the example below:

```
brooklyn.catalog:
  items:
  - id: entity-constraint-example
    itemType: entity
    name: Entity Config Example
    item:
      type: org.apache.brooklyn.entity.stock.BasicEntity
      brooklyn.parameters:
      - name: compulsoryExample
        type: string
        constraints:
        - required
      - name: addressExample
        type: string
        constraints:
        - regex: ^(?:[0-9]{1,3}\.){3}[0-9]{1,3}$
      - name: numberExample
        type: double
        constraints:
        - $brooklyn:object:
            type: org.apache.brooklyn.util.math.MathPredicates
            factoryMethod.name: greaterThan
            factoryMethod.args:
            - 0.0
        - $brooklyn:object:
            type: org.apache.brooklyn.util.math.MathPredicates
            factoryMethod.name: lessThan
            factoryMethod.args:
            - 256.0
```

An example usage of this toy example, once added to the catalog, is shown below:

```
services:
- type: entity-constraint-example
  brooklyn.config:
    compulsoryExample: foo
    addressExample: 1.1.1.1
    numberExample: 2.0
```

Inheriting Configuration

Configuration can be inherited from a super-type, and from a parent [entity](#) in the runtime management hierarchy. This applies to entities and locations. In a future release, this will be extended to also apply to policies and enrichers.

When a [blueprint](#) author defines a config key, they can explicitly specify the rules for inheritance (both for super/sub-types, and for the runtime management hierarchy). This gives great flexibility, but should be used with care so as not to surprise users of the [blueprint](#).

The default behaviour is outlined below, along with examples and details of how to explicitly define the desired behaviour.

Normal Configuration Precedence

There are several places that a configuration value can come from. If different values are specified in multiple places, then the order of precedence is as listed below:

1. Configuration on the [entity](#) itself
2. Inherited configuration from the super-type
3. Inherited configuration from the runtime type hierarchy
4. The config key's default value

Inheriting Configuration from Super-type

When using an [entity](#) from the catalog, its configuration values can be overridden. For example, consider the `entity-config-example` added to the catalog in the section [Configuration in a Catalog Item](#). We can override these values. If not overridden, then the existing values from the super-type will be used:

```
location: aws-ec2:us-east-1
services:
- type: entity-config-example
  brooklyn.config:
    custom.message: Goodbye
    launch.command: |
      echo "Sub-type launch command: $MESSAGE"
```

In this example, the `custom.message` overrides the default defined on the config key. The `launch.command` overrides the original command. The other config (e.g. `checkRunning.command`) is inherited unchanged.

It will write out: `Sub-type launch command: Goodbye` .

Inheriting Configuration from a Parent in the Runtime Management Hierarchy

Configuration passed to an [entity](#) is inherited by all child entities, unless explicitly overridden.

In the example below, the `vars.root` config key is inherited by all TomcatServer entities created under the cluster, so they will use that war:

```
services:
- type: org.apache.brooklyn.entity.group.DynamicCluster
  brooklyn.config:
    vars.root: http://search.maven.org/remotecontent?filepath=org/apache/brooklyn/example/brooklyn-example-hello-world-webapp/0.9.0/brooklyn-example-hello-world-webapp-0.9.0.war
  dynamiccluster.memberspec:
    $brooklyn:entitySpec:
```

```
type: org.apache.brooklyn.entity.webapp.tomcat.TomcatServer
```

In the above example, it would be better to have specified the `vars.root` configuration in the `TomcatServer` entity spec, rather than at the top level. This would make it clearer for the reader what is actually being configured.

The technique of inherited config can simplify some blueprints, but care should be taken. For more complex (composite) blueprints, this can be difficult to use safely; it relies on knowledge of the internals of the child components. For example, the inherited config may impact multiple sub-components, rather than just the specific entity to be changed. This is particularly true when using complex items from the catalog, and when using common config values (e.g.

```
install.version
```

). An alternative approach is to declare the expected configuration options at the top level of the catalog item, and then (within the catalog item) explicitly inject those values into the correct sub-components. Users of this catalog item would set only those exposed config options, rather than trying to inject config directly into the nested entities.

DSL Evaluation of Inherited Config

When writing blueprints that rely on inheritance from the runtime management hierarchy, it is important to understand how config keys that use DSL will be evaluated. In particular, when evaluating a DSL expression, it will be done in the context of the entity declaring the config value (rather than on the entity using the config value).

For example, consider the config value `$brooklyn:attributeWhenReady("host.name")` declared on entity X, and inherited by child entity Y. If entity Y uses this config value, it will get the "host.name" attribute of entity X.

Below is another (contrived!) example of this DSL evaluation. When evaluating `refExampleConfig`, it retrieves the value of `exampleConfig` which is the DSL expression, and evaluates this in the context of the parent entity that declares it.

Therefore `$brooklyn:config("ownConfig")` returns the parent's `ownConfig` value, and the final result for `refExampleConfig` is set to "parentValue":

```
services:
- type: org.apache.brooklyn.entity.stock.BasicApplication
  brooklyn.config:
    ownConfig: parentValue
    exampleConfig: $brooklyn:config("ownConfig")

  brooklyn.children:
  - type: org.apache.brooklyn.entity.stock.BasicEntity
    brooklyn.config:
      ownConfig: childValue
      refExampleConfig: $brooklyn:config("exampleConfig")
```

However, the web-console also shows other misleading (incorrect!) config values for the child entity. It shows the inherited config value of `exampleConfig` as "childValue" (because the REST api did not evaluate the DSL in the correct context, when retrieving the value! See <https://issues.apache.org/jira/browse/BROOKLYN-455>).

Merging Configuration Values

For some configuration values, the most logical behaviour is to merge the configuration value with that in the super-type. This depends on the type and meaning of the config key, and is thus an option when defining the config key.

Currently it is only supported for merging config keys of type Map.

Some common config keys will default to merging the values from the super-type. These config keys include those below. The value is merged with that of its super-type (but will not be merged with the value on a parent entity):

- `shell.env` : a map of environment variables to pass to the runtime shell
- `files.preinstall` : a mapping of files, to be copied before install, to destination name relative to `installDir`
- `templates.preinstall` : a mapping of templates, to be filled in and copied before pre-install, to destination name relative to `installDir`
- `files.install` : a mapping of files, to be copied before install, to destination name relative to `installDir`
- `templates.install` : a mapping of templates, to be filled in and copied before install, to destination name relative to `installDir`
- `files.runtime` : a mapping of files, to be copied before customisation, to destination name relative to `runDir`
- `templates.runtime` : a mapping of templates, to be filled in and copied before customisation, to destination name relative to `runDir`
- `provisioning.properties` : custom properties to be passed in when provisioning a new machine

A simple example of merging `shell.env` is shown below (building on the `entity-config-example` in the section [Configuration in a Catalog Item](#)). The environment variables will include the `MESSAGE` set in the super-type and the `MESSAGE2` set here:

```
location: aws-ec2:us-east-1
services:
- type: entity-config-example
  brooklyn.config:
    shell.env:
      MESSAGE2: Goodbye
    launch.command: |
      echo "Different example launch command: $MESSAGE and $MESSAGE2"
```

To explicitly remove a value from the super-type's map (rather than adding to it), a blank entry can be defined.

Entity provisioning.properties: Overriding and Merging

An [entity](#) (which extends `softwareProcess`) can define a map of `provisioning.properties`. If the [entity](#) then provisions a [location](#), it passes this map of properties to the [location](#) for obtaining the machine. These properties will override and augment the configuration on the [location](#) itself.

When deploying to a jclouds [location](#), one can specify `templateOptions` (of type map). Rather than overriding, these will be merged with any `templateOptions` defined on the [location](#).

In the example below, the VM will be provisioned with minimum 2G ram and minimum 2 cores. It will also use the merged template options value of `{placementGroup: myPlacementGroup, securityGroupIds: sg-000c3a6a}`:

```
location:
  aws-ec2:us-east-1:
    minRam: 2G
    templateOptions:
      placementGroup: myPlacementGroup
services:
- type: org.apache.brooklyn.entity.machine.MachineEntity
  brooklyn.config:
    provisioning.properties:
      minCores: 2
      templateOptions:
        securityGroupIds: sg-000c3a6a
```

The merging of `templateOptions` is shallow (i.e. maps within the `templateOptions` are not merged). In the example below, the `userMetadata` value within `templateOptions` will be overridden by the [entity](#)'s value, rather than the maps being merged; the value used when provisioning will be `{key2: val2}`:

```
location:
```

```
aws-ec2:us-east-1:
  templateOptions:
    userMetadata:
      key1: val1
  services:
  - type: org.apache.brooklyn.entity.machine.MachineEntity
    brooklyn.config:
      provisioning.properties:
        userMetadata:
          key2: val2
```

Re-inherited Versus not Re-inherited

For some configuration values, the most logical behaviour is for an **entity** to "consume" the config key's value, and thus not pass it down to children in the runtime type hierarchy. This is called "not re-inherited".

Some common config keys that will not re-inherited include:

- `install.command` (and the `pre.install.command` and `post.install.command`)
- `customize.command` (and the `pre.customize.command` and `post.customize.command`)
- `launch.command` (and the `pre.launch.command` and `post.launch.command`)
- `checkRunning.command`
- `stop.command`
- The similar commands for `VanillaWindowsProcess` powershell.
- The file and template install config keys (e.g. `files.preinstall` , `templates.preinstall` , etc)

An example is shown below. Here, the "logstash-child" is a sub-type of `VanillaSoftwareProcess` , and is co-located on the same VM as Tomcat. We don't want the Tomcat's configuration, such as `install.command` , to be inherited by the logstash child. If it was inherited, the logstash-child **entity** might re-execute the Tomcat's install command! Instead, the `install.command` config is "consumed" by the Tomcat instance and is not re-inherited:

```
services:
- type: org.apache.brooklyn.entity.webapp.tomcat.Tomcat8Server
  brooklyn.config:
    children.startable.mode: background_late
  brooklyn.children:
  - type: logstash-child
    brooklyn.config:
      logstash.elasticsearch.host: $brooklyn:entity("es").attributeWhenReady("urls.http.withBrackets")
  ...
```

"Not re-inherited" differs from "never inherited". The example below illustrates the difference, though this use is discouraged (it is mostly for backwards compatibility). The `post.install.command` is not consumed by the `BasicApplication` , so will be inherited by the `Tomcat8Server` which will consume it. The config value will therefore not be inherited by the `logstash-child` .

```
services:
- type: org.apache.brooklyn.entity.stock.BasicApplication
  brooklyn.config:
    post.install.command: echo "My post.install command"
  brooklyn.children:
  - type: org.apache.brooklyn.entity.webapp.tomcat.Tomcat8Server
    brooklyn.config:
      children.startable.mode: background_late
    brooklyn.children:
    - type: logstash-child
      brooklyn.config:
        logstash.elasticsearch.host: $brooklyn:entity("es").attributeWhenReady("urls.http.withBrackets")
  ...
```

Never Inherited

For some configuration values, the most logical behaviour is for the value to never be inherited in the runtime management hierarchy.

Some common config keys that will never be inherited include:

- `defaultDisplayName` : this is the name to use for the `entity`, if an explicit name is not supplied. This is particularly useful when adding an `entity` in a catalog item (so if the user does not give a name, it will get a sensible default). It would not be intuitive for all the children of that `entity` to also get that default name.
- `id` : the id of an `entity` (as supplied in the `YAML`, to allow references to that `entity`) is not inherited. It is the id of that specific `entity`, so must not be shared by all its children.

Inheritance Modes: Deep Dive

The javadoc in the code is useful for anyone who wants to go deep! See

`org.apache.brooklyn.config.BasicConfigInheritance` and `org.apache.brooklyn.config.ConfigInheritances` in the repo <https://github.com/apache/brooklyn-server>.

When defining a new config key, the exact semantics for inheritance can be defined. There are separate options to control config inheritance from the super-type, and config inheritance from the parent in the runtime management hierarchy.

The possible modes are:

- `NEVER_INHERITED` : indicates that a key's value should never be inherited (even if defined on an `entity` that does not know the key). Most usages will prefer `NOT_REINHERITED` .
- `NOT_REINHERITED` : indicates that a config key value (if used) should not be passed down to children / sub-types. Unlike `NEVER_INHERITED` , these values can be passed down if they are not used by the `entity` (i.e. if the `entity` does not expect it). However, when used by a child, it will not be passed down any further. If the inheritor also defines a value the parent's value is ignored irrespective (as in `OVERWRITE` ; see `NOT_REINHERITED_ELSE_DEEP_MERGE` if merging is desired).
- `NOT_REINHERITED_ELSE_DEEP_MERGE` : as `NOT_REINHERITED` but in cases where a value is inherited because a parent did not recognize it, if the inheritor also defines a value the two values should be merged.
- `OVERWRITE` : indicates that if a key has a value at both an ancestor and a descendant, the descendant and his descendants will prefer the value at the descendant.
- `DEEP_MERGE` : indicates that if a key has a value at both an ancestor and a descendant, the descendant and his descendants should attempt to merge the values. If the values are not mergable, behaviour is undefined (and often the descendant's value will simply overwrite).

Explicit Inheritance Modes

The `YAML` support for explicitly defining the inheritance mode is still work-in-progress. The options documented below will be enhanced in a future version of Brooklyn, to better support the modes described above.

In a `YAML blueprint`, within the `brooklyn.parameters` section for declaring new config keys, one can set the mode for `inheritance.type` and `inheritance.parent` (i.e. for inheritance from the super-type, and inheritance in the runtime management hierarchy). The possible values are:

- `deep_merge` : the inherited and the given value should be merged; maps within the map will also be merged
- `always` : the inherited value should be used, unless explicitly overridden by the `entity`

- `none` : the value should not be inherited; if there is no explicit value on the `entity` then the default value will be used

Below is a (contrived!) example of inheriting the `example.map` config key. When using this `entity` in a `blueprint`, the `entity`'s config will be merged with that defined in the super-type, and the parent `entity`'s value will never be inherited:

```
brooklyn.catalog:
  items:
  - id: entity-config-inheritance-example
    version: "1.1.0-SNAPSHOT"
    itemType: entity
    name: Entity Config Inheritance Example
    item:
      type: org.apache.brooklyn.entity.machine.MachineEntity
      brooklyn.parameters:
      - name: example.map
        type: java.util.Map
        inheritance.type: deep_merge
        inheritance.parent: none
        default:
          MESSAGE_IN_DEFAULT: InDefault
      brooklyn.config:
        example.map:
          MESSAGE: Hello
```

The blueprints below demonstrate the various permutations for setting configuration for the config `example.map`. This can be inspected by looking at the `entity`'s config. The config we see for `app1` is the inherited `{MESSAGE: "Hello"}`; in `app2` we define additional configuration, which will be merged to give `{MESSAGE: "Hello", MESSAGE_IN_CHILD: "InChild"}`; in `app3`, the config from the parent is not inherited because there is an explicit `inheritance.parent` of `"none"`, so it just has the value `{MESSAGE: "Hello"}`; in `app4` again the parent's config is ignored, with the super-type and `entity`'s config being merged to give `{MESSAGE: "Hello", MESSAGE_IN_CHILD: "InChild"}`.

```
location: aws-ec2:us-east-1
services:
- type: org.apache.brooklyn.entity.stock.BasicApplication
  name: app1
  brooklyn.children:
  - type: entity-config-inheritance-example

- type: org.apache.brooklyn.entity.stock.BasicApplication
  name: app2
  brooklyn.children:
  - type: entity-config-inheritance-example
    brooklyn.config:
      example.map:
        MESSAGE_IN_CHILD: InChild

- type: org.apache.brooklyn.entity.stock.BasicApplication
  name: app3
  brooklyn.config:
    example.map:
      MESSAGE_IN_PARENT: InParent
  brooklyn.children:
  - type: entity-config-inheritance-example

- type: org.apache.brooklyn.entity.stock.BasicApplication
  name: app4
  brooklyn.config:
    example.map:
      MESSAGE_IN_PARENT: InParent
  brooklyn.children:
  - type: entity-config-inheritance-example
    brooklyn.config:
      example.map:
        MESSAGE_IN_CHILD: InChild
```

A limitations of `inheritance.parent` is when inheriting values from parent and grandparent entities: a value specified on the parent will override (rather than be merged with) the value on the grandparent.

Merging Policy and Enricher Configuration Values

A current limitation is that sub-type inheritance is not supported for configuration of policies and enrichers. The current behaviour is that config is not inherited. The concept of inheritance from the runtime management hierarchy does not apply for policies and enrichers (they do not have "parents"; they are attached to an `entity`).

Setting Locations

Brooklyn supports a very wide range of target locations. With deep integration to [Apache jclouds](#), most well-known clouds and cloud platforms are supported. See the [Locations guide](#) for details and more examples.

Cloud Example

The following example is for Amazon EC2:

```
name: simple-appserver-with-location
location:
  jclouds:aws-ec2:
    region: us-east-1
    identity: AKA_YOUR_ACCESS_KEY_ID
    credential: <access-key-hex-digits>
services:
- type: org.apache.brooklyn.entity.webapp.tomcat.Tomcat8Server
```

(You'll need to replace the `identity` and `credential` with the "Access Key ID" and "Secret Access Key" for your account, as configured in the [AWS Console](#).)

Other popular public clouds include `softlayer`, `google-compute-engine`, and `rackspace-cloudservers-us`. Private cloud systems including `openstack-nova` and `cloudstack` are also supported, although for these you'll supply an `endpoint`: `https://9.9.9.9:9999/v2.0/` (or `client/api/` in the case of CloudStack) instead of the `region`.

"Bring Your Own Nodes" (BYON) Example

You can also specify pre-existing servers to use -- "bring-your-own-nodes". The example below shows a pool of machines that will be used by the entities within the application.

```
name: simple-appserver-with-location-byon
location:
  byon:
    user: brooklyn
    privateKeyFile: ~/.ssh/brooklyn.pem
    hosts:
      - 192.168.0.18
      - 192.168.0.19
services:
- type: org.apache.brooklyn.entity.webapp.tomcat.Tomcat8Server
```

Single Line and Multi Line Locations

A simple [location](#) can be specified on a single line. Alternatively, it can be split to have one configuration option per line (recommended for all but the simplest locations).

For example, the two examples below are equivalent:

```
location: byon(name="my_loc",hosts="1.2.3.4",user="bob",privateKeyFile=~/.ssh/bob_id_rsa)
```

```
location:
  byon:
```

```
name: "my_loc"
hosts:
- "1.2.3.4"
user: "bob"
privateKeyFile: "~/.ssh/bob_id_rsa"
```

Specific Locations for Specific Entities

One can define specific locations on specific entities within the `blueprint` (instead of, or as well as, defining the `location` at the top-level of the `blueprint`).

The example below will deploy Tomcat and JBoss App Server to different Bring Your Own Nodes locations:

```
name: simple-appserver-with-location-per-entity
services:
- type: org.apache.brooklyn.entity.webapp.tomcat.Tomcat8Server
  location:
    byon(hosts="192.168.0.18", user="brooklyn", privateKeyFile=~/.ssh/brooklyn.pem)
- type: org.apache.brooklyn.entity.webapp.jboss.JBoss7Server
  location:
    byon(hosts="192.168.0.19", user="brooklyn", privateKeyFile=~/.ssh/brooklyn.pem)
```

The rules for precedence when defining a `location` for an `entity` are:

- The `location` defined on that specific `entity`.
- If no `location` is defined, then the first ancestor that defines an explicit `location`.
- If still no `location` is defined, then the `location` defined at the top-level of the `blueprint`.

This means, for example, that if you define an explicit `location` on a cluster then it will be used for all members of that cluster.

Multiple Locations

Some entities are written to expect a set of locations. For example, a `DynamicFabric` will create a member `entity` in each `location` that it is given. To supply multiple locations, simply use `locations` with a `yaml` list.

In the example below, it will create a cluster of app-servers in each `location`. One `location` is used for each `dynamiccluster` ; all app-servers inside that cluster will obtain a machine from that given `location`.

```
name: fabric-of-app-server-clusters
locations:
- aws-ec2:us-east-1
- aws-ec2:us-west-1
services:
- type: org.apache.brooklyn.entity.group.DynamicFabric
  brooklyn.config:
    dynamiccluster.memberspec:
      $brooklyn:entitySpec:
        type: org.apache.brooklyn.entity.group.DynamicCluster
        brooklyn.config:
          cluster.initial.size: 3
          dynamiccluster.memberspec:
            $brooklyn:entitySpec:
              type: org.apache.brooklyn.entity.webapp.tomcat.Tomcat8Server
```

The `entity` hierarchy at runtime will have a `DynamicFabric` with two children, each of type `DynamicCluster` (each running in different locations), each of which initially has three app-servers.

For brevity, this example excludes the credentials for aws-ec2. These could either be specified in-line or defined as named locations in the catalog (see below).

Adding Locations to the Catalog

The examples above have given all the [location](#) details within the application [blueprint](#). It is also possible (and indeed preferred) to add the [location](#) definitions to the catalog so that they can be referenced by name in any [blueprint](#).

For more information see the [Operations: Catalog](#) section of the User Guide.

Externalized Configuration

For simplicity, the examples above have included the cloud credentials. For a production system, it is strongly recommended to use [Externalized Configuration](#) to retrieve the credentials from a secure credentials store, such as [Vault](#).

Use of provisioning.properties

An [entity](#) that represents a "software process" can use the configuration option `provisioning.properties` to augment the [location](#)'s configuration. For more information, see [Entity Configuration](#) details.

Configuring VMs

Another simple [blueprint](#) will just create a VM which you can use, without any software installed upon it:

```
name: simple-vm
services:
- type: org.apache.brooklyn.entity.software.base.EmptySoftwareProcess
  name: VM
  brooklyn.config:
    provisioning.properties:
      minRam: 8192mb
      minCores: 4
      minDisk: 100gb
```

We've omitted the `location` section here and in many of the examples below; add the appropriate choice when you paste your [YAML](#). Note that the `provisioning.properties` will be ignored if deploying to `localhost` or `byon` fixed-IP machines.

This will create a VM with the specified parameters in your choice of cloud. In the GUI (and in the REST API), the [entity](#) is called "VM", and the hostname and IP address(es) are reported as [sensors](#). There are many more `provisioning.properties` supported here, including:

- a `user` to create (if not specified it creates the same username as `brooklyn` is running under)
- a `password` for him or a `publicKeyFile` and `privateKeyFile` (defaulting to keys in `~/.ssh/id_rsa{.pub,}` and no password, so if you have keys set up you can immediately ssh in!)
- `machineCreateAttempts` (for dodgy clouds, and they nearly all fail occasionally!)
- and things like `imageId` and `userMetadata` and disk and networking options (e.g. `autoAssignFloatingIp` for private clouds)

For more information, see [Operations: Locations](#).

Multiple Services and Dependency Injection

We've seen the configuration of machines and how to build up clusters. Now let's return to our app-server example and explore how more interesting services can be configured, composed, and combined.

Service Configuration

We'll begin by using more key-value pairs to configure the JBoss server to run a real app:

```
name: appserver-configured
services:
- type: org.apache.brooklyn.entity.webapp.jboss.JBoss7Server
  brooklyn.config:
    wars.root: http://search.maven.org/remotecontent?filepath=org/apache/brooklyn/example/brooklyn-example-hello-world-sql-webapp/0.8.0-incubating/brooklyn-example-hello-world-sql-webapp-0.8.0-incubating.war
    http.port: 8080
```

(As before, you'll need to add the `location` info; `localhost` will work for these and subsequent examples.)

When this is deployed, you can see management information in the Brooklyn Web Console, including a link to the deployed application (downloaded to the target machine from the `hello-world` URL), running on port 8080.

Tip: If port 8080 might be in use, you can specify `8080+` to take the first available port \geq 8080; the actual port will be reported as a `sensor` by Brooklyn.

Multiple Services

If you explored the `hello-world-sql` application we just deployed, you'll have noticed it tries to access a database. And it fails, because we have not set one up. Let's do that now:

```
name: appserver-w-db
services:
- type: org.apache.brooklyn.entity.webapp.jboss.JBoss7Server
  name: AppServer HelloWorld
  brooklyn.config:
    wars.root: http://search.maven.org/remotecontent?filepath=org/apache/brooklyn/example/brooklyn-example-hello-world-sql-webapp/0.8.0-incubating/brooklyn-example-hello-world-sql-webapp-0.8.0-incubating.war
    http.port: 8080+
  java.sysprops:
    brooklyn.example.db.url:
      $brooklyn:formatString:
        - jdbc:%s?user=%s\&password=%s
        - $brooklyn:component("db").attributeWhenReady("datastore.url")
        - visitors
        - brooklyn
        - $brooklyn:external("brooklyn-demo-sample", "hidden-brooklyn-password")
- type: org.apache.brooklyn.entity.database.mysql.MySqlNode
  id: db
  name: DB HelloWorld Visitors
  brooklyn.config:
    creation.script.password: $brooklyn:external("brooklyn-demo-sample", "hidden-brooklyn-password")
    datastore.creation.script.url: https://github.com/apache/brooklyn-library/raw/master/examples/simple-web-cluster/src/main/resources/visitors-creation-script.sql
```

Here there are a few things going on:

- We've added a second service, which will be the database; you'll note the database has been configured to run a

custom setup script

- We've injected the URL of the second service into the appserver as a Java system property (so our app knows where to find the database)
- We've used externalized config to keep secret information out of the `blueprint`; this is loaded at runtime from an `externalized config provider`, such as a remote credentials store

Caution: Be careful if you write your `YAML` in an editor which attempts to put "smart-quotes" in. All quote characters must be plain ASCII, not fancy left-double-quotes and right-double-quotes!

There are as many ways to do dependency injection as there are developers, it sometimes seems; our aim in Brooklyn is not to say this has to be done one way, but to support the various mechanisms people might need, for whatever reasons. (We each have our opinions about what works well, of course; the one thing we do want to call out is that being able to dynamically update the injection is useful in a modern agile application -- so we are definitively **not** recommending this Java system property approach ... but it is an easy one to demo!)

The way the dependency injection works is again by using the `$brooklyn:` DSL, this time referring to the `component("db")` (looked up by the `id` field on our DB component), and then to a `sensor` emitted by that component. All the database entities emit a `database.url` `sensor` when they are up and running; the `attributeWhenReady` DSL method will store a pointer to that `sensor` (a Java Future under the covers) in the Java system properties map which the JBoss `entity` reads at launch time, blocking if needed.

This means that the deployment occurs in parallel, and if the database comes up first, there is no blocking; but if the JBoss `entity` completes its installation and downloading the WAR, it will wait for the database before it launches. At that point the URL is injected, first passing it through `formatString` to include the credentials for the database (which are defined in the database creation script).

An Aside: Substitutability

Don't like JBoss? Is there something about Maria? One of the modular principles we follow in Brooklyn is substitutability: in many cases, the config keys, sensors, and effectors are defined in superclasses and are portable across multiple implementations.

Here's an example deploying the same application but with different flavors of the components:

```
name: appserver-w-db-other-flavor
services:
- type: org.apache.brooklyn.entity.webapp.tomcat.TomcatServer
  name: AppServer HelloWorld
  brooklyn.config:
    wars.root: http://search.maven.org/remotecontent?filepath=org/apache/brooklyn/example/brooklyn-example-hello-world-sql-webapp/0.8.0-incubating/brooklyn-example-hello-world-sql-webapp-0.8.0-incubating.war
    http.port: 8080+
    java.sysprops:
      brooklyn.example.db.url:
        $brooklyn:formatString:
          - jdbc:%s?s?user=%s\&password=%s
          - $brooklyn:component("db").attributeWhenReady("datastore.url")
          - visitors
          - brooklyn
          - $brooklyn:external("brooklyn-demo-sample", "hidden-brooklyn-password")
- type: org.apache.brooklyn.entity.database.mariadb.MariaDbNode
  id: db
  name: DB HelloWorld Visitors
  brooklyn.config:
    creation.script.password: $brooklyn:external("brooklyn-demo-sample", "hidden-brooklyn-password")
    datastore.creation.script.url: https://github.com/apache/brooklyn-library/raw/master/examples/simple-web-cl
    uster/src/main/resources/visitors-creation-script.sql
    provisioning.properties:
      minRam: 8192
```

By changing two lines we've switched from JBoss and MySQL to Tomcat and MariaDB.

We've also brought in the `provisioning.properties` from the VM example earlier so our database has 8GB RAM. Any of those properties, including `imageId` and `user`, can be defined on a per-entity basis.

Custom Entities

So far we've covered how to configure and compose entities. There's a large library of blueprints available, but there are also times when you'll want to write your own.

For complex use cases, you can write JVM, but for many common situations, some of the highly-configurable blueprints make it easy to write in [YAML](#), including `bash` and `Chef`.

Vanilla Software using `bash`

The following [blueprint](#) shows how a simple script can be embedded in the [YAML](#) (the `|` character is special [YAML](#) which makes it easier to insert multi-line text):

```
name: Simple Netcat Server Example
location: localhost
services:
- type: org.apache.brooklyn.entity.software.base.VanillaSoftwareProcess
  name: Simple Netcat Server
  brooklyn.config:
    launch.command: |
      echo hello | nc -l 4321 &
      echo $! > $PID_FILE
```

This starts a simple `nc` listener on port 4321 which will respond `hello` to the first session which connects to it. Test it by running `telnet localhost 4321` or opening `http://localhost:4321` in a browser.

Note that it only allows you connect once, and after that it fails. This is deliberate! We'll repair this later in this example. Until then however, in the *Applications* view you can click the server, go to the `Effectors` tab, and click `restart` to bring it back to life.

This is just a simple script, but it shows how any script can be easily embedded here, including a script to download and run other artifacts. Many artifacts are already packaged such that they can be downloaded and launched with a simple script, and `vanillaSoftwareProcess` can also be used for them.

Downloading Files

We can specify a `download.url` which downloads an artifact (and automatically unpacking TAR, TGZ, and ZIP archives) before running `launch.command` relative to where that file is installed (or unpacked), with the default `launch.command` being `./start.sh`.

So if we create a file `/tmp/netcat-server.tgz` containing just `start.sh` in the root which contains the line `echo hello | nc -l 4321`, we can instead write our example as:

```
name: Simple Netcat Example From File
location: localhost
services:
- type: org.apache.brooklyn.entity.software.base.VanillaSoftwareProcess
  name: Simple Netcat Server
  brooklyn.config:
    download.url: file:///tmp/netcat-server.tgz
    launch.command: |
      ./start.sh &
      echo $! > $PID_FILE
```

Determining Successful Launch

The default method used to determine a successful launch of `VanillaSoftwareProcess` is to run a command over ssh to do a health check. The health check is done post-launch (repeating until it succeeds, before then reporting that the `entity` has started).

The default command used to carry out this health check will determine if the pid, written to `$PID_FILE` is running. This is why we included in the `entity`'s launch script the line `echo $! > $PID_FILE`.

You'll observe this if you connect to one of the netcat services (e.g. via `telnet localhost 4321`): the `nc` process exits afterwards, causing Brooklyn to set the `entity` to an `ON_FIRE` state. (You can also test this with a `killall nc`).

There are other options for determining health: you can set `checkRunning.command` and `stop.command` instead, as documented on the javadoc and config keys of the `org.apache.brooklyn.entity.software.base.VanillaSoftwareProcess` class, and those scripts will be used instead of checking and stopping the process whose PID is in `$PID_FILE`. For example:

```
name: Netcat Example with Explicit Check and Stop Commands
location: localhost
services:
- type: org.apache.brooklyn.entity.software.base.VanillaSoftwareProcess
  name: Simple Netcat Server
  brooklyn.config:
    launch.command: |
      echo hello | nc -l 4321 &
      echo $! > $PID_FILE

    # The following overrides demonstrate the use of a custom shell environment as well as
    # check-running and stop commands. These are optional; default behavior will "do the
    # right thing" with the pid file automatically.

    shell.env:
      CHECK_MARKER: "checkRunning"
      STOP_MARKER: "stop"
    checkRunning.command: |
      echo $CHECK_MARKER >> DATE && test -f "$PID_FILE" && ps -p `cat $PID_FILE` >/dev/null
    stop.command: |
      echo $STOP_MARKER >> DATE && test -f "$PID_FILE" && { kill -9 `cat $PID_FILE`; rm $PID_FILE; }
```

Periodic Health Check

After start-up is complete, the health check described above is also run periodically, defaulting to every 5 seconds (configured with the config key `softwareProcess.serviceProcessIsRunningPollPeriod`).

This ssh-based polling can be turned off by configuring `sshMonitoring.enabled: false`. However, if no alternative health-check is defined then failure of the process would never be detected by Brooklyn.

See [Health Check Sensors](#) for alternative ways of detecting failures.

Port Inferencing

If you're deploying to a cloud machine, a firewall might block the port 4321. We can tell Brooklyn to open this port explicitly by specifying `inboundPorts: [4321]`; however a more idiomatic way is to specify a config ending with `.port`, such as:

```
name: Netcat Example with Port Opened
location: localhost
services:
```

```
- type: org.apache.brooklyn.entity.software.base.VanillaSoftwareProcess
  name: Simple Netcat Server

  brooklyn.config:
    # matching the regex `.*\port` will cause the port to be opened
    # if in a cloud where configurable security groups are available
    netcat.port: 4321

    launch.command: |
      echo hello | nc -l 4321 &
      echo $! > $PID_FILE
```

The regex for ports to be opened can be configured using the config `inboundPorts.configRegex` (which has `.*\port` as the default value).

Config keys of type `org.apache.brooklyn.api.location.PortRange` (aka `port`) have special behaviour: when configuring, you can use range notation `8000-8100` or `8000+` to tell Brooklyn to find **one** port matching; this is useful when ports might be in use. In addition, any such config key will be opened, irrespective of whether it matches the `inboundPorts.configRegex` . To prevent any inferencing of ports to open, you can set the config `inboundPorts.autoInfer` to `false` .

Furthermore, the port inferencing capability takes in account static `configKey` fields that are defined on any `Entity` sub-class. So, `configKey` fields that are based on `PortRanges` type will be also included as required open ports.

Note that in the example above, `netcat.port` must be specified in a `brooklyn.config` block. This block can be used to hold any config (including for example the `launch.command`), but for convenience Brooklyn allows config keys declared on the underlying type to be specified up one level, alongside the type. However config keys which are *not* declared on the type *must* be declared in the `brooklyn.config` block.

Passing custom variables

`Blueprint` scripts can be parametrised through environment variables, making them reusable in different use-cases. Define the variables in the `env` block and then reference them using the standard bash notation:

```
name: Netcat Example with Environment Vars
location: localhost
services:
- type: org.apache.brooklyn.entity.software.base.VanillaSoftwareProcess
  name: Simple Netcat Server

  brooklyn.config:
    launch.command: |
      echo $MESSAGE | nc -l $NETCAT_PORT &
      echo $! > $PID_FILE

    shell.env:
      MESSAGE: hello
      NETCAT_PORT: 4321
```

Non-string objects in the `env` map will be serialized to JSON before passing them to the script.

Declaring New Config Keys

We can define config keys to be presented to the user using the `brooklyn.parameters` block:

```
name: Netcat Example with Parameter
location: localhost
services:
```

```
- type: org.apache.brooklyn.entity.software.base.VanillaSoftwareProcess
name: Simple Netcat Server
brooklyn.config:
  launch.command: |
    echo $MESSAGE | nc -l $NETCAT_PORT &
    echo $! > $PID_FILE

  shell.env:
    MESSAGE: $brooklyn:config("message")
    NETCAT_PORT: $brooklyn:attributeWhenReady("netcat.port")

brooklyn.parameters:
- name: message
  description: a message to send to the caller
  default: hello
- name: netcat.port
  type: port
  description: the port netcat should run on
  default: 4321+
```

The example above will allow a user to specify a message to send back and the port where netcat will listen. The metadata on these parameters is available at runtime in the UI and through the API, and is used when populating a catalog.

The example also shows how these values can be passed as environment variables to the launch command. The `$brooklyn:config(...)` function returns the config value supplied or default. For the type `port`, an attribute `sensor` is also created to report the *actual* port used after port inference, and so the `$brooklyn:attributeWhenReady(...)` function is used. (If `$brooklyn:config("netcat.port")` had been used, `4321+` would be passed as `NETCAT_PORT`.)

This gives us quite a bit more power in writing our [blueprint](#):

- Multiple instances of the server can be launched simultaneously on the same host, as the `4321+` syntax enables Brooklyn to assign them different ports
- If this type is added to the catalog, a user can configure the message and the port; we'll show this in the next section

Using the Catalog and Clustering

The *Catalog* tab allows you to add blueprints which you can refer to in other blueprints. In that tab, click + then [YAML](#), and enter the following:

```
brooklyn.catalog:
  id: netcat-example
  version: "1.0"
  itemType: entity
  item:
    type: org.apache.brooklyn.entity.software.base.VanillaSoftwareProcess
    name: Simple Netcat Server

    brooklyn.config:
      launch.command: |
        echo $MESSAGE | nc -l $NETCAT_PORT &
        echo $! > $PID_FILE

      shell.env:
        MESSAGE: $brooklyn:config("message")
        NETCAT_PORT: $brooklyn:attributeWhenReady("netcat.port")

    brooklyn.parameters:
      - name: message
        description: a message to send to the caller
        default: hello
      - name: netcat.port
```



```

type: port
description: the port netcat should run on
default: 4321+

brooklyn.enrichers:
- type: org.apache.brooklyn.enricher.stock.Transformer
brooklyn.config:
  uniqueTag: main-uri-generator
  enricher.sourceSensor: $brooklyn:sensor("host.address")
  enricher.targetSensor: $brooklyn:sensor("main.uri")
  enricher.targetValue:
    $brooklyn:formatString:
      - "http://%s:%s/"
      - $brooklyn:attributeWhenReady("host.address")
      - $brooklyn:attributeWhenReady("netcat.port")

```

This is the same example as in the previous section, wrapped according to the catalog [YAML](#) requirements, with one new block added defining an [enricher](#). An [enricher](#) creates a new [sensor](#) from other values; in this case it will create a `main.uri` [sensor](#) by populating a `printf`-style string `"http://%s:%s/"` with the [sensor](#) values.

With this added to the catalog, we can reference the type `netcat-example` when we deploy an application. Return to the *Home* or *Applications* tab, click `+`, and submit this [YAML blueprint](#):

```

name: Netcat Type Reference Example
location: localhost
services:
- type: netcat-example
  message: hello from netcat using a registered type

```

This extends the previous [blueprint](#) which we registered in the catalog, meaning that we don't need to include it each time. Here, we've elected to supply our own message, but we'll use the default port. More importantly, we can package it for others to consume -- or take items others have built.

We can go further and use this to deploy a cluster, this time giving a custom port as well as a custom message:

```

name: Netcat Cluster Example
location: localhost
services:
- type: org.apache.brooklyn.entity.group.DynamicCluster
  brooklyn.config:
    dynamiccluster.memberspec:
      $brooklyn:entitySpec:
        type: netcat-example
        message: hello from cluster member
        netcat.port: 8000+
    cluster.initial.size: 3
    dynamiccluster.restartMode: parallel

```

In either of the above examples, if you explore the tree in the *Applications* view and look at the *Summary* tab of any of the server instances, you'll now see the URL where netcat is running. But remember, netcat will stop after one run, so you'll only be able to use each link once before you have to restart it. You can also run `restart` on the cluster, and if you haven't yet experimented with `resize` on the cluster you might want to do that.

Attaching Policies

Besides detecting this failure, Brooklyn policies can be added to the [YAML](#) to take appropriate action. A simple recovery here might just to automatically restart the process:

```

name: Netcat Example with Restarter Policy

```

```
location: localhost
services:
- type: org.apache.brooklyn.entity.software.base.VanillaSoftwareProcess
  id: netcat-server
  name: Simple Netcat Server
  brooklyn.config:
    launch.command: |
      echo hello | nc -l 4321 &
      echo $! > $PID_FILE
  brooklyn.enrichers:
- type: org.apache.brooklyn.policy.ha.ServiceFailureDetector
  brooklyn.config:
    # wait 15s after service fails before propagating failure
    serviceFailedStabilizationDelay: 15s
  brooklyn.policies:
- type: org.apache.brooklyn.policy.ha.ServiceRestarter
  brooklyn.config:
    # repeated failures in a time window can cause the restarter to abort,
    # propagating the failure; a time window of 0 will mean it always restarts!
    failOnRecurringFailuresInThisDuration: 0
```

Autonomic management in Brooklyn often follows the principle that complex behaviours emerge from composing simple policies. The **blueprint** above uses one **policy** to triggering a failure **sensor** when the service is down, and another responds to such failures by restarting the service. This makes it easy to configure various aspects, such as to delay to see if the service itself recovers (which here we've set to 15 seconds) or to bail out on multiple failures within a time window (which again we are not doing). Running with this **blueprint**, you'll see that the service shows as on fire for 15s after a `telnet localhost 4321`, before the **policy** restarts it.

Sensors and Effectors

Effectors

For an even more interesting way to test it, look at the **blueprint** defining a **netcat server and client**. This uses `brooklyn.initializers` to define an **effector** to `sayHiNetcat` on the `Simple Pinger` client, using `env` variables to inject the `netcat-server` **location** and `parameters` to pass in per-**effector** data:

```
env:
  TARGET_HOSTNAME: $brooklyn:entity("netcat-server").attributeWhenReady("host.name")
brooklyn.initializers:
- type: org.apache.brooklyn.core.effector.ssh.SshCommandEffector
  brooklyn.config:
    name: sayHiNetcat
    description: Echo a small hello string to the netcat entity
    command: |
      echo $message | nc $TARGET_HOSTNAME 4321
    parameters:
      message:
        description: The string to pass to netcat
        defaultValue: hi netcat
```

Sensors

This **blueprint** also uses initializers to define sensors on the `netcat-server` **entity** so that the `$message` we passed above gets logged and reported back:

```
launch.command: |
  echo hello | nc -l 4321 >> server-input &
  echo $! > $PID_FILE
```

```
brooklyn.initializers:
- type: org.apache.brooklyn.core.sensor.ssh.SshCommandSensor
brooklyn.config:
  name: output.last
  period: 1s
  command: tail -1 server-input
```

Windows Command Sensor

Like the [blueprint](#) above, the following example also uses `brooklyn.initializers` to define sensors on the `entity`, this time however it is a windows VM and uses `WinRmCommandSensor`.

```
- type: org.apache.brooklyn.entity.software.base.VanillaWindowsProcess
brooklyn.config:
  launch.command: echo launching
  checkRunning.command: echo running
brooklyn.initializers:
- type: org.apache.brooklyn.core.sensor.windows.WinRmCommandSensor
brooklyn.config:
  name: ip.config
  period: 60s
  command: hostname
```

Health Check Sensors

As mentioned [previously](#), the default health check is to execute the check-running command over ssh every 5 seconds. This can be very CPU intensive when there are many entities. An alternative is to disable the ssh-polling (by setting `sshMonitoring.enabled: false`) and to configure a different health-check.

See documentation on the [Entity's error status](#) for how Brooklyn models an `entity`'s health.

In the snippet below, we'll define a new health-check `sensor` (via http polling), and will automatically add this to the `service.notUp.indicators`. If that map is non-empty, then the `entity`'s `service.isUp` will be set automatically to `false`:

```
services:
- type: org.apache.brooklyn.entity.software.base.VanillaSoftwareProcess
brooklyn.config:
  launch.command: |
    ...
  checkRunning.command: true
  sshMonitoring.enabled: false

brooklyn.initializers:
- type: org.apache.brooklyn.core.sensor.http.HttpRequestSensor
brooklyn.config:
  name: http.healthy
  period: 5s
  suppressDuplicates: true
  jsonPath: "$"
  uri:
    $brooklyn:formatString:
    - "http://%s:8080/healthy"
    - $brooklyn:attributeWhenReady("host.name")

brooklyn.enrichers:
- type: org.apache.brooklyn.enricher.stock.UpdatingMap
brooklyn.config:
  enricher.sourceSensor: $brooklyn:sensor("http.healthy")
  enricher.targetSensor: $brooklyn:sensor("service.notUp.indicators")
  enricher.updatingMap.computing:
    $brooklyn:object:
      type: "com.google.guava:com.google.common.base.Functions"
```

```
factoryMethod.name: "forMap"
factoryMethod.args:
- true: null
  false: "false"
- "no value"
```

The `HttpRequestSensor` configures the `entity` to poll every 5 seconds on the given URI, taking the json result as the `sensor` value.

The `updatingMap enricher` uses that `sensor` to populate an entry in the `service.notup.indicators`. It transforms the `http.healthy sensor` value using the given function: if the http poll returned `true`, then it is mapped to `null` (so is removed from the `service.noup.indicators`); if the poll returned `false`, then `"false"` is added to the indicators map; otherwise `"no value"` is added to the indicators map.

Summary

These examples do relatively simple things, but they illustrate many of the building blocks used in real-world blueprints, and how they can often be easily described and combined in Brooklyn `YAML` blueprints.

Catalog

Apache Brooklyn provides a **catalog**, which is a persisted collection of versioned blueprints and other resources. A set of blueprints is loaded from the `default.catalog.bom` in the Brooklyn folder by default and additional ones can be added through the web console or CLI. Blueprints in the catalog can be deployed directly, via the Brooklyn CLI or the web console, or referenced in other blueprints using their `id`.

- [Catalog Items YAML Syntax](#)
- [Templates and the Add-Application Wizard](#)
- [Versioning](#)
- [Catalog Management](#)
- [Bundling](#)
- [Brooklyn Server Command Line Arguments](#)

Catalog Items YAML Syntax

Catalog Items YAML Syntax

An item or items to be added to the catalog is defined by a **YAML** file, specifying the catalog metadata for the items and the actual **blueprint** or resource definition.

General YAML Schema

Catalog items can be defined using the general structure below:

```
brooklyn.catalog:
  <catalog-metadata>
  items:
    - <additional-catalog-metadata>
      item:
        <blueprint-or-resource-definition>
    - <additional-catalog-metadata>
      item:
        <blueprint-or-resource-definition>
```

Alternatively, a single catalog item can be defined using the following general structure:

```
brooklyn.catalog:
  <catalog-metadata>
  item:
    <blueprint-or-resource-definition>
```

For example, the **YAML** below adds to the catalog a Tomcat **entity** with some additional default configuration:

```
brooklyn.catalog:
  items:
    - id: tomcat-server
      version: "1.0.0"
      itemType: entity
      item:
        type: org.apache.brooklyn.entity.webapp.tomcat.Tomcat8Server
        brooklyn.config:
          webapp.enabledProtocols: https
          httpsSsl:
            url: classpath://org/apache/brooklyn/entity/webapp/sample-java-keystore.jks
            alias: myname
            password: mypass
```

Catalog Metadata

Catalog metadata fields supply the additional information required in order to register an item in the catalog. These fields can be supplied as `key: value` entries where either the `<catalog-metadata>` Or `<additional-catalog-metadata>` placeholders are, with the latter overriding the former unless otherwise specified below.

The following metadata is *required* for all items:

- `id` : a human-friendly unique identifier for how this catalog item will be referenced from blueprints
- `version` : multiple versions of a **blueprint** can be installed and used simultaneously; this field disambiguates between

blueprints of the same `id`. Note that this is typically *not* the version of the software being installed, but rather the version of the `blueprint`. For more information on versioning, see below. (Also note `YAML` treats numbers differently to Strings. Explicit quotes are recommended, to avoid `1.10` being interpreted as the number `1.1`.)

- `itemType` : the type of the item being defined. The supported item types are:
 - `entity`
 - `template`
 - `policy`
 - `location`

To reference a catalog item in another `blueprint`, simply reference its ID and optionally its version number. For instance, if we've added an item with metadata `{ id: datastore, version: "1.0" }` (such as the example below), we could refer to it in another `blueprint` with:

```
services:
- type: datastore:1.0
```

In addition to the above fields, exactly **one** of the following is also required:

- `item` : the `YAML` for an `entity`, or `policy`, or `location` specification (a map containing `type` and optional `brooklyn.config`). For a "template" item, it should be a map containing `services` (i.e. the usual `YAML` format for a full application `blueprint`). **Or**
- `items` : a list of catalog items, where each entry in the map follows the same schema as the `brooklyn.catalog` value, and the keys in these map override any metadata specified as a sibling of this `items` key (or, in the case of `brooklyn.libraries` they add to the list); if there are references between items, then order is important: `items` are processed in order, depth-first, and forward references are not supported. Entries can be URL to another catalog file to include, inheriting the metadata from the current hierarchy. Libraries defined so far in the metadata will be used to load classpath entries. For example:

```
brooklyn.catalog:
brooklyn.libraries:
- http://some.server.or.other/path/my.jar
items:
- classpath://my-catalog-entries-inside-jar.bom
- some-property: value
  include: classpath://more-catalog-entries-inside-jar.bom
- id: use-from-my-catalog
  version: "1.0.0"
  itemType: entity
  item:
    type: some-type-defined-in-my-catalog-entries
    brooklyn.config:
      some.config: "some value"
```

The following optional catalog metadata is supported:

- `name` : a nicely formatted display name for the item, used when presenting it in a GUI.
- `description` : supplies an extended textual description for the item.
- `iconUrl` : points to an icon for the item, used when presenting it in a GUI. The URL prefix `classpath` is supported but these URLs may *not* refer to resources in any OSGi bundle in the `brooklyn.libraries` section (to prevent requiring all OSGi bundles to be loaded at launch). Icons are instead typically installed either at the web server from which the OSGi bundles or catalog items are supplied or in the `conf` folder of the Brooklyn distro.
- `scanJavaAnnotations` [experimental; deprecated]: if provided (as `true`), this will scan any locally provided library URLs for types annotated `@Catalog` and extract metadata to include them as catalog items. If no libraries are specified this will scan the default classpath. This feature will likely be removed. Also note that external OSGi dependencies are not supported and other metadata (such as versions, etc) may not be applied.
- `brooklyn.libraries` : a list of pointers to OSGi bundles required for the catalog item. This can be omitted if

blueprints are pure **YAML** and everything required is included in the classpath and catalog. Where custom Java code or bundled resources is needed, however, OSGi JARs supply a convenient packaging format and a very powerful versioning format. Libraries should be supplied in the form `brooklyn.libraries: ["http://...", "http://..."]`, or as `brooklyn.libraries: [{ name: symbolic-name, version: "1.0", url: http://... }, ...]` if `symbolic-name:1.0` might already be installed from a different URL and you want to skip the download. Note that these URLs should point at immutable OSGi bundles; if the contents at any of these URLs changes, the behaviour of the **blueprint** may change whenever a bundle is reloaded in a Brooklyn server, and if entities have been deployed against that version, their behavior may change in subtle or potentially incompatible ways. To avoid this situation, it is highly recommended to use OSGi version stamps as part of the URL.

- `include` : A URL to another catalog file to include, inheriting the meta from the current hierarchy. Libraries defined so far in the meta will be used to load classpath entries. `include` must be used when you have sibling properties. If it's the only property it may be skipped by having the URL as the value - see `items` example above.

Catalog YAML Examples

A Simple Example

The following example installs the `RiakNode` **entity**, making it also available as an application template, with a nice display name, description, and icon. It can be referred in other blueprints to as `datastore:1.0`, and its implementation will be the Java class `org.apache.brooklyn.entity.nosql.riak.RiakNode` included with Brooklyn.

```
brooklyn.catalog:
  id: datastore
  version: "1.0"
  itemType: template
  iconUrl: classpath://org/apache/brooklyn/entity/nosql/riak/riak.png
  name: Datastore (Riak)
  description: Riak is an open-source NoSQL key-value data store.
  item:
    services:
      - type: org.apache.brooklyn.entity.nosql.riak.RiakNode
        name: Riak Node
```

Multiple Items

This **YAML** will install three items:

```
brooklyn.catalog:
  version: "1.1"
  iconUrl: classpath://org/apache/brooklyn/entity/nosql/riak/riak.png
  description: Riak is an open-source NoSQL key-value data store.
  items:
    - id: riak-node
      itemType: entity
      item:
        type: org.apache.brooklyn.entity.nosql.riak.RiakNode
        name: Riak Node
    - id: riak-cluster
      itemType: entity
      item:
        type: org.apache.brooklyn.entity.nosql.riak.RiakCluster
        name: Riak Cluster
    - id: datastore
      name: Datastore (Riak Cluster)
      itemType: template
      item:
        services:
          - type: riak-cluster
        brooklyn.config:
          # the default size is 3 but this can be changed to suit your requirements
```



```

initial.size: 3
provisioning.properties:
  # you can also define machine specs
  minRam: 8gb

```

The items this will add to the catalog are:

- `riak-node` , as before, but with a different name
- `riak-cluster` as a convenience short name for the `org.apache.brooklyn.entity.nosql.riak.RiakCluster` class
- `datastore` , now pointing at the `riak-cluster` [blueprint](#), in SoftLayer and with the given size and machine spec, as the default implementation for anyone requesting a `datastore` (and if installed atop the previous example, new references to `datastore` will access this version because it is a higher number); because it is a template, users will have the opportunity to edit the [YAML](#) (see below). (This must be supplied after `riak-cluster` , because it refers to `riak-cluster` .)

Locations in the Catalog

In addition to blueprints, locations can be added to the Apache Brooklyn catalog. The example below shows a [location](#) for the vagrant configuration used in the [getting started guide](#), formatted as a catalog entry.

```

brooklyn.catalog:
  id: vagrant
  version: "1.0"
  itemType: location
  name: Vagrant getting started location
  item:
    type: byon
    brooklyn.config:
      user: vagrant
      password: vagrant
      hosts:
        - 10.10.10.101
        - 10.10.10.102
        - 10.10.10.103
        - 10.10.10.104

```

Once this has been added to the catalog it can be used as a named [location](#) in [yaml](#) blueprints using:

```
location: vagrant
```

Legacy Syntax

The following legacy and experimental syntax is also supported, but deprecated:

```

<blueprint-definition>
brooklyn.catalog:
  <catalog-metadata>

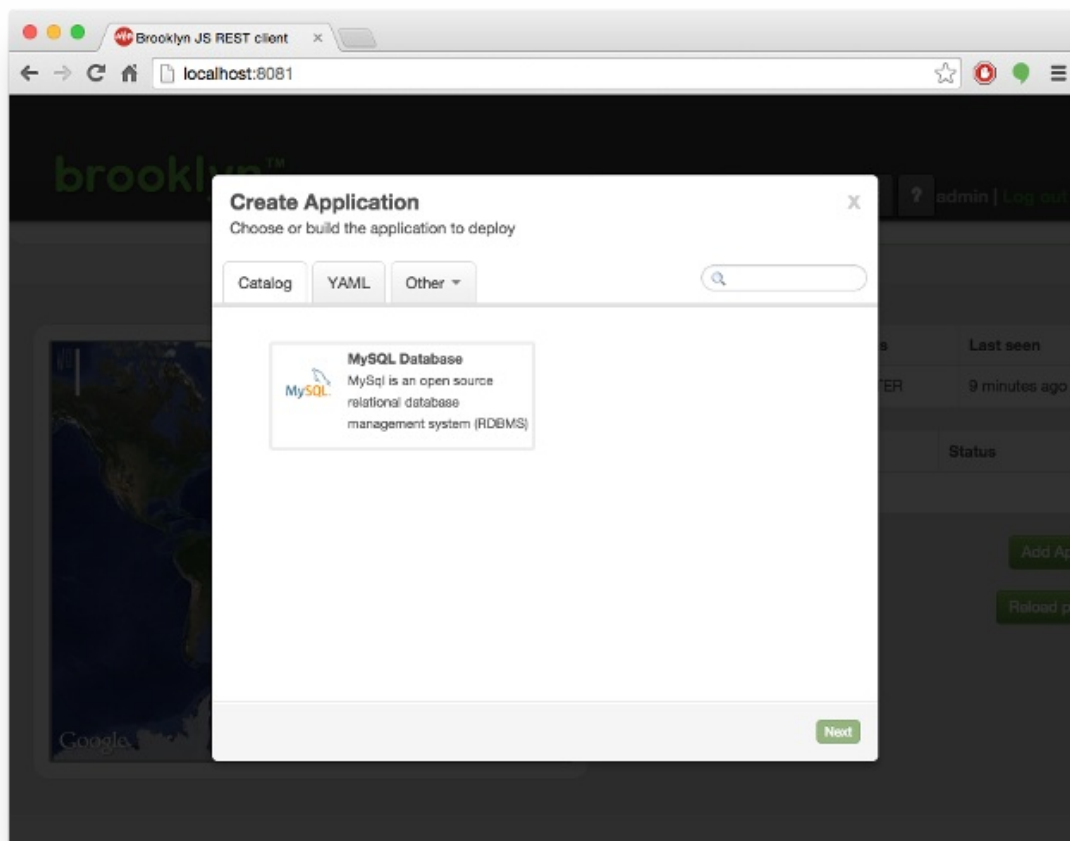
```

In this format, the `brooklyn.catalog` block is optional; and an `id` in the `<blueprint-definition>` will be used to determine the catalog ID. This is primarily supplied for OASIS CAMP 1.1 compatibility, where the same [YAML blueprint](#) can be POSTed to the catalog endpoint to add to a catalog or POSTed to the applications endpoint to deploy an instance. (This syntax is discouraged as the latter usage, POSTing to the applications endpoint, will ignored the `brooklyn.catalog` information; this means references to any `item` blocks in the `<catalog-metadata>` will not be resolved, and any OSGi `brooklyn.libraries` defined there will not be loaded.)

Templates and the Add-Application Wizard

Templates and the Add-Application Wizard

A `template` is a full application. It consists of one or more entities inside an application (though this is also composable: it can be used as part of another application). When a `template` is added to the catalog, the `blueprint` will appear in the 'Create Application' dialog as shown here:



Versioning

Brooklyn supports multiple versions of a type to be installed and used at the same time. Versions are a first-class concept and are often prominently displayed in the UI.

In order to do this, Brooklyn requires that the `id:version` string be unique across the catalog: it is normally an error to add a type if a type with the same `id:version` is present. The exceptions to this are if the definition is identical, or if the `version` is noted as a `SNAPSHOT`. In extraordinary circumstances it may be appropriate to delete a given `id:version` definition and then add the new one, but this is discouraged and the usual practice is to:

- Use a `-SNAPSHOT` qualifier suffix on your version when developing
- Increase the version number when making a change to a non-SNAPSHOT type

When adding to the catalog, if no version is supplied, Brooklyn will typically use `0.0.0-SNAPSHOT`, and some clients will automatically create and increment the version number for the catalog item.

When deploying a [blueprint](#), if a version number is not specified Brooklyn will typically use the highest ordered version (see "Ordering" below) in the catalog for the referenced type, and will thereafter lock the use of that version in that [blueprint](#). (An exception is where types are co-bundled or an explicit search path is defined; in the context of evaluating one type, Brooklyn may prefer versions in the same bundle or on the search path.)

Versioning Syntax

Version numbers in Brooklyn are recommended to follow the following syntax:

```
<major> ( "." <minor> ( "." <patch> )? )? ( "-" <qualifier> )?
```

where the `<major>`, `<minor>`, and `<patch>` parts are numbers in accordance with [semver](#) semantic versioning, assumed to be `0` if omitted, and an `<qualifier>` is made up of letters, numbers, `"-"` and `"_"` in accordance with [OSGi](#) (see sections 1.3.2 and 3.2.5).

Examples:

- `1.2`
- `2.0.0`
- `3`
- `2.0.0-SNAPSHOT`
- `1.10-rc3-20170619`

Snapshots and Ordering

The string `SNAPSHOT` appearing anywhere in the version indicates a pre-release version; if this string is not present the version is treated as a release version.

When taking an ordering, for instance to find the highest version, snapshot versions are always considered lower than release versions. Next, the natural order is taken on the major, minor, and patch fields. Next, a version with no qualifier is considered higher than one with a qualifier. Finally, the qualifier is taken in natural order.

The natural order here is defined as numeric order for digit sequences (`"9" < "10"`) and ASCII-lexicographic comparison elsewhere (`"a" < "b"`), which is normally what people will expect for versions (`1.9 < 1.10` and `"1.1-rc9-b" < "1.1-rc10-a"`).

Thus the *order* of the list of examples above is:

- `2.0.0-SNAPSHOT`
- `1.2`
- `1.10-rc3-20170619`
- `2.0.0`
- `3`

For most practical purposes, `3`, `3.0`, and `3.0.0` are treated as equivalent, but if referencing a version you should use the exact version string defined. The version `3.0.0` is different, as the `-0` indicates a qualifier, and is ordered before a `3.0.0`.

Advanced: Other Version Syntaxes

Other version syntaxes are supported with the following restrictions:

- Version strings MUST NOT contain a colon character (`:`)
- Version strings MUST NOT be empty
- Fragments that do not follow the recommended syntax may be ignored when determining version uniqueness (e.g. adding both `"1.0.0-v1.1"` and `"1.0.0-v11"` can result in one bundle *_replacing* the other rather than both versions being loaded)

This means in practice that almost any common version scheme can be used. However the recommended scheme will fit more neatly alongside types from other sources.

Internally when installing bundles, Brooklyn needs to produce OSGi-compliant versions. For the recommended syntax, this mapping consists of replacing the first occurrence of `"-"` with `"."` and setting `0` values for absent minor and patch versions. Thus when looking at the OSGi view, instead of version `1.10-rc3-20170619` you will see `1.10.0.rc3-20170619`. Apart from the omission of `0` as minor and patch versions, this mapping is guaranteed to be one-to-one so no conflicts will occur if the recommended syntax is used. Bundles `foo:3`, `foo:3.0`, and `foo:3.0.0` would all be installed using OSGi version `3.0.0`, and so would conflict and block installation if there is any change (and replace if they have a `-SNAPSHOT` qualifier); references to bundles can use `3` or `3.0` or `3.0.0`, though as noted above types contained within would have to be referenced using the exact version string supplied. (If different versions are specified on individual types than for the bundle itself -- which is not recommended -- then the conversion to OSGi does not apply, and the versions are not treated as equal; in such edge cases the ordering obeys numeric then ASCII ordering on segments, so we have `3 < 3.0 < 3.01 < 3.1 < 3.09 < 3.9 < 3.10` and `v-1 < v.1 < v_1`.)

If not using the recommended syntax, the mapping proceeds by treating the first dot-separated fragment as the qualifier and converts unsupported characters in a qualifier to an underscore; thus `1.x` becomes `1.0.0.x`, `v1` becomes `0.0.0.v1`, and `"1.0.0-v1.1"` becomes `"1.0.0.v1_1"` hence the bundle replacement noted above.

If you are creating an OSGi `MANIFEST.MF` for a bundle that also contains a `catalog.bom`, you will need to use the mapped result (OSGi version syntax) in the manifest, but should continue to use the Brooklyn-recommended syntax in the `catalog.bom`.

For those who are curious, the reason for the Brooklyn version syntax is to reconcile the popular usage of semver and maven with the internal requirement to use OSGi versions. Semver, OSGi, and maven conventions agree on up to three numeric dot-separated tokens, but differ quite significantly afterwards, with Brooklyn adopting what seems to be the most popular choices in each.

A summary of the main differences between Brooklyn and other versioning syntaxes is as follows:

- Qualifier preceded by hyphen (maven and semver semantics, different to OSGi which wants a dot)
- Underscores allowed in qualifiers (OSGi and maven semantics, different to semver)
- Periods and plus not allowed in qualifiers (OSGi semantics and maven convention, different to semver which gives them special meaning)
- The ordering used in Brooklyn is different to that used in OSGi (where qualifiers come after the unqualified version)

and don't do a numeric comparison)

- `SNAPSHOT` treated specially (maven semantics)
- Maven's internal to-OSGi conversion is different for some non-recommended syntax strings (e.g. `10rc1` becomes `10.0.0.rc1` in Brooklyn but Maven will map it by default to `0.0.0.10rc1`)

Catalog Management

Catalog Management

The Catalog tab in the web console will show all versions of catalog items, and allow you to add new items.

Adding to the Catalog

On the UI the "add" button  at the top of the menu panel allows the addition of new Applications to the catalog, via [YAML](#), and of new Locations.

In addition to the GUI, items can be added to the catalog via the REST API with a `POST` of the [YAML](#) file to `/v1/catalog` endpoint. To do this using `curl` :

```
curl -u admin:password http://127.0.0.1:8081/v1/catalog --data-binary @/path/to/riak.catalog.bom
```

Or using the CLI:

```
br catalog add /path/to/riak.catalog.bom
```

Deleting from the Catalog

On the UI, if an item is selected, a 'Delete' button in the detail panel can be used to delete it from the catalog.

Using the REST API, you can delete a versioned item from the catalog using the corresponding endpoint. For example, to delete the item with id `datastore` and version `1.0` with `curl` :

```
curl -u admin:password -X DELETE http://127.0.0.1:8081/v1/catalog/applications/datastore/1.0
```

Note: Catalog items should not be deleted if there are running apps which were created using the same item. During rebinding the catalog item is used to reconstruct the [entity](#).

If you have running apps which were created using the item you wish to delete, you should instead deprecate the catalog item. Deprecated catalog items will not appear in the add application wizard, or in the catalog list but will still be available to Brooklyn for rebinding. The option to display deprecated catalog items in the catalog list will be added in a future release.

Deprecation applies to a specific version of a catalog item, so the full id including the version number is passed to the REST API as follows:

```
curl -u admin:password -X POST http://127.0.0.1:8081/v1/catalog/entities/MySQL:1.0/deprecated/true
```

Bundling

Bundling Catalog Resources

It is possible to upload catalog items and associated resources as a single bundle to AMP. This is useful when you have a [blueprint](#) that needs to reference external scripts, icons, config files or other resources, or when you have multiple blueprints that you want to keep in sync. Brooklyn will persist any uploaded bundles so that they are available after a restart or on HA failover.

The bundle must be a ZIP file including a `catalog.bom` in the root. (The `br` CLI will create a ZIP from a local folder, for convenience.) The `catalog.bom` must declare a `bundle` identifier and a `version`, following Brooklyn's [versioning](#) rules. Brooklyn will keep track of that bundle, allowing items to be added and removed as a group, and associated resources to be versioned and included alongside them. With SNAPSHOT-version bundles, it allows replacement of multiple related items at the same time, and in advanced cases it allows setting up dependent bundles (specified as `brooklyn.libraries` or, for people familiar with OSGi, the `Required-Bundle` manifest header) which will be searched if a [blueprint](#) in one bundle references resources from another bundle.

Resources in the bundle can be referenced from the `catalog.bom` by using the `classpath:` URL protocol, as in `classpath://path/to/script.sh`. This can also be used to load resources in explicitly declared dependent bundles.

Example

In this example, we will create a simple `my-server` catalog item, bundled with a simple script. The script will be run when launching the server.

First, create a folder called `bundleFolder`, then add a file called `myfile.sh` to it. The contents of `myfile.sh` should be as follows:

```
echo Hello, World!
```

Now create a file in `bundleFolder` called `catalog.bom` with the following contents:

```
brooklyn.catalog:
  bundle: MyServerBundle
  version: 1.0.0
  items:
    - id: my-server
      item:
        type: org.apache.brooklyn.entity.software.base.VanillaSoftwareProcess
        brooklyn.config:
          files.runtime:
            classpath://myfile.sh: files/myfile.sh
          launch.command: |
            chmod +x ./files/myfile.sh
            ./files/myfile.sh
          checkRunning.command: echo "Running"
```

The `bundle: MyServerBundle` line specifies the OSGi bundle name for this bundle. Any resources included in this bundle will be accessible on the classpath, but will be scoped to this bundle. This prevents an issue where multiple bundles include the same resource.

To create the bundle, simply use the `br` command as follows. This will create a zip and send it to Brooklyn. Please note you can also specify a zip file (either on the file system or hosted remotely):


```
br catalog add bundleFolder
```

This will have added our bundle to the catalog. We can now deploy an instance of our server as follows. Please note that in this example we deploy to localhost. If you have not setup your machine to use localhost please see the instructions [here](#) or use a non localhost [location](#).

```
location: localhost
services:
- type: my-server
```

We can now see the result of running that script. In the UI find the activities for this application. The start activity has a sub task called launch (you will have to click through multiple activities called start/launch. Looking at the stdout of the launch task you should see:

```
Hello, World!
```

Alternatively you can view the script directly if you run the following against localhost. Please note that brooklyn-username and the id of your app will be different.

```
cat /tmp/brooklyn-username/apps/nl9djqbq2i/entities/VanillaSoftwareProcess_g52gahfxnt/files/myfile.sh
```

It should look like this:

```
echo Hello, World!
```

Now modify `myfile.sh` to contain a different message, change the version number in `catalog.bom` to `1.1.0`, and use the `br` command to send the bundle to the server.

If you now deploy a new instance of the server using the same [YAML](#) as above, you should be able to confirm that the new script has been run (either by looking at the stdout of the launch task, or looking at the script itself)

At this point, it is also possible to deploy the original `Hello, World!` version by explicitly stating the version number in the [YAML](#):

```
location: localhost
services:
- type: my-server:1.0.0
```

To demonstrate the scoping, you can create another bundle with the following `catalog.bom`. Note the bundle name and [entity](#) id have been changed, but it still references a script with the same name.

```
brooklyn.catalog:
  bundle: DifferentServerBundle
  version: 1.0.0
  item:
    id: different-server
    type: org.apache.brooklyn.entity.software.base.VanillaSoftwareProcess
    brooklyn.config:
      files.runtime:
        classpath://myfile.sh: files/myfile.sh
      launch.command: |
        chmod +x ./files/myfile.sh
        ./files/myfile.sh

    checkRunning.command:
      echo "Running"
```

Now create a new `myfile.sh` script with a different message, and use the `br` command to send the bundle to Brooklyn.

Now deploy a `blueprint` which deploys all three servers. Each of the three deployments will utilise the script that was included with their bundle.

```
location: localhost
services:
- type: my-server:1.0.0
- type: my-server:1.1.0
- type: different-server
```

Note: All three entities copy a file from `classpath://myfile.sh`, but as they are in different bundles, the scripts copied to the server will be different.

Brooklyn Server Command Line Arguments

Brooklyn Server Command Line Arguments

The command line arguments when launching `brooklyn` include several commands for working with the catalog.

- `--catalogAdd <file.bom>` will add the catalog items in the `bom` file
- `--catalogReset` will reset the catalog to the initial state (based on `brooklyn/default.catalog.bom` on the classpath, by default in a dist in the `conf/` directory)
- `--catalogInitial <file.bom>` will set the catalog items to use on first run, on a catalog reset, or if persistence is off

If `--catalogInitial` is not specified, the default initial catalog at `brooklyn/default.catalog.bom` will be used. As `scanJavaAnnotations: true` is set in `default.catalog.bom`, Brooklyn will scan the classpath for catalog items, which will be added to the catalog. To launch Brooklyn without initializing the catalog, use `--catalogInitial classpath://brooklyn/empty.catalog.bom`

If `persistence` is enabled, catalog additions will remain between runs. If items that were previously added based on items in `brooklyn/default.catalog.bom` or `--catalogInitial` are deleted, they will not be re-added on subsequent restarts of brooklyn. I.e. `--catalogInitial` is ignored if persistence is enabled and persistent state has already been created.

For more information on these commands, run `brooklyn help launch`.

Clusters, Specs, and Composition

What if you want multiple machines?

One way is just to repeat the `- type: org.apache.brooklyn.entity.software.base.EmptySoftwareProcess` block, but there's another way which will keep your powder **DRY**:

```
name: cluster-vm
services:
- type: org.apache.brooklyn.entity.group.DynamicCluster
  brooklyn.config:
    cluster.initial.size: 5
    dynamiccluster.memberspec:
      $brooklyn:entitySpec:
        type: org.apache.brooklyn.entity.software.base.EmptySoftwareProcess
        name: VM
        provisioning.properties:
          minRam: 8g
          minCores: 4
          minDisk: 100g
```

Here we've composed the previous **blueprint** introducing some new important concepts, the `DynamicCluster` the `$brooklyn` DSL, and the "entity-spec". Let's unpack these.

The `dynamicCluster` creates a set of homogeneous instances. At design-time, you specify an initial size and the specification for the **entity** it should create. At runtime you can restart and stop these instances as a group (on the `dynamicCluster`) or refer to them individually. You can resize the cluster, attach enrichers which aggregate sensors across the cluster, and attach policies which, for example, replace failed members or resize the cluster dynamically.

The specification is defined in the `dynamiccluster.memberspec` key. As you can see it looks very much like the previous **blueprint**, with one extra line. Entries in the **blueprint** which start with `$brooklyn:` refer to the Brooklyn DSL and allow a small amount of logic to be embedded (if there's a lot of logic, it's recommended to write a **blueprint** YAML plugin or write the **blueprint** itself as a plugin, in Java or a JVM-supported language).

In this case we want to indicate that the parameter to `dynamiccluster.memberspec` is an **entity** specification (`EntitySpec` in the underlying type system); the `entityspec` DSL command will do this for us. The example above thus gives us 5 VMs identical to the one we created in the previous section.

Enrichers

Enrichers provide advanced manipulation of an [entity's sensor](#) values. See below for documentation of the stock enrichers available in Apache Brooklyn.

Transformer

```
org.apache.brooklyn.enricher.stock.Transformer
```

Takes a source [sensor](#) and modifies it in some way before publishing the result in a new [sensor](#). See below an example using `$brooklyn:formatString`.

```
brooklyn.enrichers:
- type: org.apache.brooklyn.enricher.stock.Transformer
  brooklyn.config:
    enricher.sourceSensor: $brooklyn:sensor("urls.tcp.string")
    enricher.targetSensor: $brooklyn:sensor("urls.tcp.withBrackets")
    enricher.targetValue: $brooklyn:formatString("[%s]", $brooklyn:attributeWhenReady("urls.tcp.string"))
```

Propagator

```
org.apache.brooklyn.enricher.stock.Propagator
```

Use propagator to duplicate one [sensor](#) as another, giving the supplied [sensor](#) mapping. The other use of Propagator is where you specify a producer (using `$brooklyn:entity(...)` as below) from which to take sensors; in that mode you can specify `propagate` as a list of sensors whose names are unchanged, instead of (or in addition to) this map.

```
brooklyn.enrichers:
- type: org.apache.brooklyn.enricher.stock.Propagator
  brooklyn.config:
    enricher.producer: $brooklyn:entity("cluster")
- type: org.apache.brooklyn.enricher.stock.Propagator
  brooklyn.config:
    sensorMapping:
      $brooklyn:sensor("url"): $brooklyn:sensor("org.apache.brooklyn.core.entity.Attributes", "main.uri")
```

Custom Aggregating

```
org.apache.brooklyn.enricher.stock.Aggregator
```

Aggregates multiple [sensor](#) values (usually across a tier, esp. a cluster) and performs a supplied aggregation method to them to return an aggregate figure, e.g. sum, mean, median, etc.

```
brooklyn.enrichers:
- type: org.apache.brooklyn.enricher.stock.Aggregator
  brooklyn.config:
    enricher.sourceSensor: $brooklyn:sensor("webapp.reqs.perSec.windowed")
    enricher.targetSensor: $brooklyn:sensor("webapp.reqs.perSec.perNode")
    enricher.aggregating.fromMembers: true
    transformation: average
```

There are a number of additional configuration keys available for the Aggregators:

Configuration Key	Default	Description

<code>enricher.transformation.untyped</code>	list	Specifies a transformation, as a function from a collection to the value, or as a string matching a pre-defined named transformation, such as 'average' (for numbers), 'sum' (for numbers), 'isQuorate' (to compute a quorum), 'first' (the first value, or null if empty), or 'list' (the default, putting any collection of items into a list)
<code>quorum.check.type</code>		The requirement to be considered quorate -- possible values: 'all', 'allAndAtLeastOne', 'atLeastOne', 'atLeastOneUnlessEmpty', 'alwaysHealthy', 'allAndAtLeastOne'
<code>quorum.total.size</code>	1	The total size to consider when determining if quorate

Joiner

`org.apache.brooklyn.enricher.stock.Joiner`

Joins a `sensor` whose output is a list into a single item joined by a separator.

```
brooklyn.enrichers:
- type: org.apache.brooklyn.enricher.stock.Joiner
  brooklyn.config:
    enricher.sourceSensor: $brooklyn:sensor("urls.tcp.list")
    enricher.targetSensor: $brooklyn:sensor("urls.tcp.string")
    uniqueTag: urls.quoted.string
```

There are a number of additional configuration keys available for the joiner:

Configuration Key	Default	Description
<code>enricher.joiner.separator</code>	,	Separator string to insert between each argument
<code>enricher.joiner.keyValueSeparator</code>	=	Separator string to insert between each key-value pair
<code>enricher.joiner.joinMapEntries</code>	false	Whether to add map entries as key-value pairs or just use the value
<code>enricher.joiner.quote</code>	true	Whether to bash-escape each parameter and wrap in double-quotes
<code>enricher.joiner.minimum</code>	0	Minimum number of elements to join; if fewer than this, sets null
<code>enricher.joiner.maximum</code>	null	Maximum number of elements to join (null means all elements taken)

Delta Enricher

`org.apache.brooklyn.policy.enricher.DeltaEnricher`

Converts an absolute `sensor` into a delta `sensor` (i.e. the difference between the current and previous value)

Time-weighted Delta

`org.apache.brooklyn.enricher.stock.YamlTimeWeightedDeltaEnricher`

Converts absolute `sensor` values into a difference over time. The `enricher.delta.period` indicates the measurement interval.

```
brooklyn.enrichers:
- type: org.apache.brooklyn.enricher.stock.YamlTimeWeightedDeltaEnricher
```

```
brooklyn.config:
  enricher.sourceSensor: reqs.count
  enricher.targetSensor: reqs.per_sec
  enricher.delta.period: 1s
```

Rolling Mean

```
org.apache.brooklyn.policy.enricher.RollingMeanEnricher
```

Transforms a [sensor](#) into a rolling average based on a fixed window size. This is useful for smoothing sample type metrics, such as latency or CPU time

Rolling Time-window Mean

```
org.apache.brooklyn.policy.enricher.RollingTimeWindowMeanEnricher
```

Transforms a [sensor](#)'s data into a rolling average based on a time window. This time window can be specified with the config key `confidenceRequired` - Minimum confidence level (ie period covered) required to publish a rolling average (default `8d`).

Http Latency Detector

```
org.apache.brooklyn.policy.enricher.RollingTimeWindowMeanEnricher.HttpLatencyDetector
```

An [Enricher](#) which computes latency in accessing a URL, normally by periodically polling that URL. This is then published in the sensors `web.request.latency.last` and `web.request.latency.windowed`.

There are a number of additional configuration keys available for the Http Latency Detector:

Configuration Key	Default	Description
latencyDetector.url		The URL to compute the latency of
latencyDetector.urlSensor		A sensor containing the URL to compute the latency of
latencyDetector.urlPostProcessing		Function applied to the urlSensor value, to determine the URL to use
latencyDetector.rollup		The window size (in duration) over which to compute
latencyDetector.requireServiceUp	false	Require the service is up
latencyDetector.period	1s	The period of polling

Combiner

```
org.apache.brooklyn.enricher.stock.Combiner
```

Can be used to combine the values of sensors. This [enricher](#) should be instantiated using

```
Enrichers.builder().combining(..) . This enricher is only available in Java blueprints and cannot be used in YAML.
```

Note On Enricher Producers

If an [entity](#) needs an [enricher](#) whose source [sensor](#) (`enricher.sourceSensor`) belongs to another [entity](#), then the [enricher](#) configuration must include an `enricher.producer` key referring to the other [entity](#).

For example, if we consider the Transformer from above, suppose that `enricher.sourceSensor: $brooklyn:sensor("urls.tcp.list")` is actually a `sensor` on a different `entity` called `load.balancer`. In this case, we would need to supply an `enricher.producer` value.

```
brooklyn.enrichers:  
- type: org.apache.brooklyn.enricher.stock.Transformer  
  brooklyn.config:  
    enricher.sourceSensor: $brooklyn:sensor("urls.tcp.string")  
    enricher.targetSensor: $brooklyn:sensor("urls.tcp.withBrackets")  
    enricher.targetValue: $brooklyn:formatString("[%s]", $brooklyn:attributeWhenReady("urls.tcp.string"))
```

It is important to note that the value supplied to `enricher.producer` must be immediately resolvable. While it would be valid DSL syntax to write:

```
enricher.producer: brooklyn:entity($brooklyn:attributeWhenReady("load.balancer.entity"))
```

(assuming the `load.balancer.entity` `sensor` returns a Brooklyn `entity`), this will not function properly because `enricher.producer` will unsuccessfully attempt to get the supplied `entity` immediately.

Policies

Policies perform the active management enabled by Brooklyn. They can subscribe to [entity](#) sensors and be triggered by them (or they can run periodically, or be triggered by external systems).

Policies can add subscriptions to sensors on any [entity](#). Normally a [policy](#) will subscribe to sensors on either its associated [entity](#), that [entity](#)'s children and/or to the members of a "group" [entity](#).

Common uses of a [policy](#) include the following:

- perform calculations,
- look up other values,
- invoke effectors (management policies) or,
- cause the [entity](#) associated with the [policy](#) to emit [sensor](#) values ([enricher](#) policies).

Entities can have zero or more `Policy` instances attached to them.

Off-the-Shelf Policies

Policies are highly reusable as their inputs, thresholds and targets are customizable. Config key details for each [policy](#) can be found in the Catalog in the Brooklyn UI.

HA/DR and Scaling Policies

AutoScaler Policy

- `org.apache.brooklyn.policy.autoscaling.AutoScalerPolicy`

Increases or decreases the size of a Resizable [entity](#) based on an aggregate [sensor](#) value, the current size of the [entity](#), and customized high/low watermarks.

An AutoScaler [policy](#) can take any [sensor](#) as a metric, have its watermarks tuned live, and target any resizable [entity](#) - be it an application server managing how many instances it handles, or a tier managing global capacity.

e.g. if the average request per second across a cluster of Tomcat servers goes over the high watermark, it will resize the cluster to bring the average back to within the watermarks.

```
brooklyn.policies:
- type: org.apache.brooklyn.policy.autoscaling.AutoScalerPolicy
  brooklyn.config:
    metric: webapp.reqs.perSec.perNode
    metricUpperBound: 3
    metricLowerBound: 1
    resizeUpStabilizationDelay: 2s
    resizeDownStabilizationDelay: 1m
    maxPoolSize: 3
```

ServiceRestarter Policy

- `org.apache.brooklyn.policy.ha.ServiceRestarter`

Attaches to a `SoftwareProcess` or to anything `Startable` which emits `ha.entityFailed` on failure (or other configurable `sensor`), and invokes `restart` on that failure. If there is a subsequent failure within a configurable time interval or if the restart fails, this gives up and emits `ha.entityFailed.restart` for other policies to act upon or for manual intervention.

```
brooklyn.policies:
- type: org.apache.brooklyn.policy.ha.ServiceRestarter
  brooklyn.config:
    failOnRecurringFailuresInThisDuration: 5m
```

Typically this is used in conjunction with the `FailureDetector enricher` to emit the trigger `sensor`. The [introduction to policies](#) shows a worked example of these working together.

ServiceReplacer Policy

- `org.apache.brooklyn.policy.ha.ServiceReplacer`

The `ServiceReplacer` attaches to a `DynamicCluster` and replaces a failed member in response to `ha.entityFailed` (or other configurable `sensor`).

The [introduction to policies](#) shows a worked example of this `policy` in use.

SshMachineFailureDetector Policy

- `org.apache.brooklyn.policy.ha.SshMachineFailureDetector`

The `SshMachineFailureDetector` is an HA `policy` for monitoring an `SshMachine`, emitting an event if the connection is lost/restored.

ConnectionFailureDetector Policy

- `org.apache.brooklyn.policy.ha.ConnectionFailureDetector`

The `ConnectionFailureDetector` is an HA `policy` for monitoring an http connection, emitting an event if the connection is lost/restored.

Optimization Policies

PeriodicEffector Policy

- `org.apache.brooklyn.policy.action.PeriodicEffectorPolicy`

The `PeriodicEffectorPolicy` calls an `effector` with a set of arguments at a specified time and date. The `policy` monitors the `sensor` configured by `start.sensor` and will only start when this is set to `true`. The default `sensor` checked is `service.isup`, so that the `policy` will not execute the `effector` until the `entity` is started. The following example calls a `resize` `effector` to resize a cluster up to 10 members at 8am and then down to 1 member at 6pm.

```
- type: org.apache.brooklyn.policy.action.PeriodicEffectorPolicy
  brooklyn.config:
    effector: resize
    args:
      desiredSize: 10
      period: 1 day
      time: 08:00:00
- type: org.apache.brooklyn.policy.action.PeriodicEffectorPolicy
  brooklyn.config:
    effector: resize
```

```
args:
  desiredSize: 1
  period: 1 day
  time: 18:00:00
```

ScheduledEffector Policy

- `org.apache.brooklyn.policy.action.ScheduledEffectorPolicy`

The `ScheduledEffectorPolicy` calls an `effector` at a specific time. The `policy` monitors the `sensor` configured by `start.sensor` and will only execute the `effector` at the specified time if this is set to `true`.

There are two modes of operation, one based solely on `policy` configuration where the `effector` will execute at the time set using the `time` key or after the duration set using the `wait` key, or by monitoring sensors. The `policy` monitors the `scheduler.invoke.now` `sensor` and will execute the `effector` immediately when its value changes to `true`. When the `scheduler.invoke.at` `sensor` changes, it will set a time in the future when the `effector` should be executed.

The following example calls a `backup` `effector` every night at midnight.

```
- type: org.apache.brooklyn.policy.action.ScheduledEffectorPolicy
  brooklyn.config:
    effector: backup
    time: 00:00:00
```

FollowTheSun Policy

- `org.apache.brooklyn.policy.followthesun.FollowTheSunPolicy`

The `FollowTheSunPolicy` is for moving work around to follow the demand. The work can be any Movable `entity`. This is currently available in `yaml` blueprints.

LoadBalancing Policy

- `org.apache.brooklyn.policy.loadbalancing.LoadBalancingPolicy`

The `LoadBalancingPolicy` is attached to a pool of "containers", each of which can host one or more migratable "items". The `policy` monitors the workrates of the items and effects migrations in an attempt to ensure that the containers are all sufficiently utilized without any of them being overloaded.

Lifecycle and User Management Policies

StopAfterDuration Policy

- `org.apache.brooklyn.policy.action.StopAfterDurationPolicy`

The `StopAfterDurationPolicy` can be used to limit the lifetime of an `entity`. After a configured time period expires the `entity` will be stopped.

ConditionalSuspend Policy

- `org.apache.brooklyn.policy.ha.ConditionalSuspendPolicy`

The ConditionalSuspendPolicy will suspend and resume a target [policy](#) based on configured suspend and resume sensors.

CreateUser Policy

- `org.apache.brooklyn.policy.jclouds.os.CreateUserPolicy`

The CreateUserPolicy Attaches to an [Entity](#) and monitors for the addition of a [location](#) to that [entity](#), the [policy](#) then adds a new user to the VM with a randomly generated password, with the SSH connection details set on the [entity](#) as the `createuser.vm.user.credentials` [sensor](#).

AdvertiseWinRMLogin Policy

- `org.apache.brooklyn.location.winrm.WinRmMachineLocation`

This is similar to the CreateUserPolicy. It will monitor the addition of WinRmMachineLocation to an [entity](#) and then create a [sensor](#) advertising the administrative user's credentials.

Writing a Policy

Your First Policy

Policies perform the active management enabled by Brooklyn. Each [policy](#) instance is associated with an [entity](#), and at runtime it will typically subscribe to sensors on that [entity](#) or children, performing some computation and optionally actions when a subscribed [sensor](#) event occurs. This action might be invoking an [effector](#) or emitting a new [sensor](#), depending the desired behavior is.

Writing a [policy](#) is straightforward. Simply extend `AbstractPolicy`, overriding the `setEntity` method to supply any subscriptions desired:

```
@Override
public void setEntity(EntityLocal entity) {
    super.setEntity(entity)
    subscribe(entity, TARGET_SENSOR, this)
}
```

and supply the computation and/or activity desired whenever that event occurs:

```
@Override
public void onEvent(SensorEvent<Integer> event) {
    int val = event.getValue()
    if (val % 2 == 1)
        entity.sayYoureOdd();
}
```

You'll want to do more complicated things, no doubt, like access other entities, perform multiple subscriptions, and emit other sensors -- and you can. See the best practices below and source code for some commonly used policies and enrichers, such as `AutoScalerPolicy` and `RollingMeanEnricher`.

One rule of thumb, to close on: try to keep policies simple, and compose them together at runtime; for instance, if a complex computation triggers an action, define one [enricher](#) [policy](#) to aggregate other sensors and emit a new [sensor](#), then write a second [policy](#) to perform that action.

Best Practice

The following recommendations should be considered when designing policies:

Management should take place as "low" as possible in the hierarchy

- place management responsibility in policies at the `entity`, as much as possible ideally management should take run as a `policy` on the relevant `entity`
- place escalated management responsibility at the parent `entity`. Where this is impractical, perhaps because two aspects of an `entity` are best handled in two different places, ensure that the separation of responsibilities is documented and there is a group membership relationship between secondary/aspect managers.

Policies should be small and composable

e.g. one `policy` which takes a `sensor` and emits a different, enriched `sensor`, and a second `policy` which responds to the enriched `sensor` of the first (e.g. a `policy` detects a process is maxed out and emits a `TOO_HOT sensor`; a second `policy` responds to this by scaling up the VM where it is running, requesting more CPU)

Where a policy cannot resolve a situation at an entity, the issue should be escalated to a manager with a compatible policy.

Typically escalation will go to the `entity` parent, and then cascade up. e.g. if the earlier VM CPU cannot be increased, the `TOO_HOT` event may go to the parent, a cluster `entity`, which attempts to balance. If the cluster cannot balance, then to another `policy` which attempts to scale out the cluster, and should the cluster be unable to scale, to a third `policy` which emits `TOO_HOT` for the cluster.

Management escalation should be carefully designed so that policies are not incompatible

Order policies carefully, and mark sensors as "handled" (or potentially "swallow" them locally), so that subsequent policies and parent entities do not take superfluous (or contradictory) corrective action.

Implementation Classes

Extend `AbstractPolicy`, or override an existing `policy`.

Effectors

Effectors perform an operation of some kind, carried out by a Brooklyn [Entity](#). They can be manually invoked or triggered by a [Policy](#).

Common uses of an [effector](#) include the following:

- Perform a command on a remote machine.
- Collect data and publish them to sensors.

Entities have default effectors, the lifecycle management effectors like `start`, `stop`, `restart`, and clearly more `Effectors` can be attached to them.

Off-the-Shelf Effectors

Effectors are highly reusable as their inputs, thresholds and targets are customizable.

SSHCommandEffector

An `Effector` to invoke a command on a node accessible via SSH.

It enables execution of a `command` in a specific `execution director` (`executionDir`) by using a custom `shell environment` (`shellEnv`). By default, the specified command will be executed on the [entity](#) where the [effector](#) is attached or on all *children* or all *members* (if it is a group) by configuring `executionTarget`.

There are a number of additional configuration keys available for the `SSHCommandEffector`:

Configuration Key	Default	Description
<code>command</code>		command to be executed on the execution target
<code>executionDir</code>		possible values: 'GET', 'HEAD', 'POST', 'PUT', 'PATCH', 'DELETE', 'OPTIONS', 'TRACE'
<code>shellEnv</code>		custom shell environment where the command is executed
<code>executionTarget</code>	ENTITY	possible values: 'MEMBERS', 'CHILDREN'

Here is a simple example of an `sshCommandEffector` definition:

```
brooklyn.initializers:  
- type: org.apache.brooklyn.core.effector.ssh.SshCommandEffector  
  brooklyn.config:  
    name: sayHiNetcat  
    description: Echo a small hello string to the netcat entity  
    command: |  
      echo $message | nc $TARGET_HOSTNAME 4321  
    parameters:  
      message:  
        description: The string to pass to netcat  
        defaultValue: hi netcat
```

See [here](#) for more details.

HTTPCommandEffector

An `Effector` to invoke HTTP endpoints.

It allows the user to specify the URI, the HTTP verb, credentials for authentication and HTTP headers.

There are a number of additional configuration keys available for the `HTTPCommandEffector` :

Configuration Key	Default	Description
uri		URI of the endpoint
httpVerb		possible values: 'GET', 'HEAD', 'POST', 'PUT', 'PATCH', 'DELETE', 'OPTIONS', 'TRACE'
httpUsername		user name for the authentication
httpPassword		password for the authentication
headers	application/json	It explicitly supports <code>application/x-www-form-urlencoded</code>
httpPayload		The body of the http request
jsonPath		A jsonPath expression to extract values from a json object
jsonPathAndSensors		A map where keys are jsonPath expressions and values the name of the <code>sensor</code> where to publish extracted values

When a the header `HttpHeaders.CONTENT_TYPE` is equals to `application/x-www-form-urlencoded` and the `httpPayload` is a `map` , the payload is transformed into a single string using `URLEncoded` .

```
brooklyn.initializers:
- type: org.apache.brooklyn.core. effector .http.HttpCommandEffector
  brooklyn.config:
    name: request-access-token
    description: Request an access token for the Azure API
    uri:
      $brooklyn:formatString:
        - "https://login.windows.net/%s/oauth2/token"
        - $brooklyn:config("tenant.id")
    httpVerb: POST
    httpPayload:
      resource: https://management.core.windows.net/
      client_id: $brooklyn:config("application.id")
      grant_type: client_credentials
      client_secret: $brooklyn:config("application.secret")
    jsonPathAndSensors:
      $.access_token: access.token
    headers:
      Content-Type: "application/x-www-form-urlencoded"
```

See [here](#) for more details.

AddChildrenEffector

An `Effector` to add a child `blueprint` to an `entity`.

```
brooklyn.initializers:
- type: org.apache.brooklyn.core. effector .AddChildrenEffector
  brooklyn.config:
    name: add_tomcat
    blueprint_yaml: |
      name: sample
```

```

description: Tomcat sample JSP and servlet application.
origin: http://www.oracle.com/nCAMP/Hand
services:
-
  type: io.camp.mock:AppServer
  name: Hello WAR
  wars:
    /: hello.war
  controller.spec:
    port: 80

brooklyn.catalog:
name: catalog-name
type: io.camp.mock.MyApplication
version: 0.9
libraries:
- name: org.apache.brooklyn.test.resources.osgi.brooklyn-test-osgi-entities
  version: 0.1.0
  url: classpath:/brooklyn/osgi/brooklyn-test-osgi-entities.jar
auto_start: true

```

One of the config keys `BLUEPRINT_YAML` (containing a [YAML blueprint](#) (map or string)) or `BLUEPRINT_TYPE` (containing a string referring to a catalog type) should be supplied, but not both.

See [here](#) for more details.

Writing an Effector

Your First Effector

Effectors generally perform actions on entities. Each [effector](#) instance is associated with an [entity](#), and at runtime it will typically execute an operation, collect the result and, potentially, publish it as [sensor](#) on that [entity](#), performing some computation.

Writing an [effector](#) is straightforward. Simply extend `AddEffector`, providing an implementation for `newEffectorBuilder` and adding a constructor that consumes the builder or override an existing [effector](#).

```

public MyEffector(ConfigBag params) {
    super(newEffectorBuilder(params).build());
}

public static EffectorBuilder<String> newEffectorBuilder(ConfigBag params) {
    EffectorBuilder<String> eff = AddEffector.newEffectorBuilder(String.class, params);
    eff.impl(new Body(eff.buildAbstract(), params));
    return eff;
}

```

and supply an `EffectorBody` similar to:

```

protected static class Body extends EffectorBody<String> {
    ...

    @Override
    public String call(final ConfigBag params) {
        ...
    }
}

```


Best Practice

The following recommendations should be considered when designing effectors:

Effectors should be small and composable

One **effector** which executes a command and emits a **sensor**, and a second **effector** which uses the previous **sensor**, if defined, to execute another operation.

Clusters and Policies

Now let's bring the concept of the "cluster" back in. We could wrap our appserver in the same `dynamiccluster` we used earlier, although then we'd need to define and configure the load balancer. But another `blueprint`, the `ControlledDynamicWebAppCluster`, does this for us. It takes the same `dynamiccluster.memberspec`, so we can build a fully functional elastic 3-tier deployment of our `hello-world-sql` application as follows:

```
name: appserver-clustered-w-db
services:
- type: org.apache.brooklyn.entity.webapp.ControlledDynamicWebAppCluster
  brooklyn.config:
    cluster.initial.size: 2
    dynamiccluster.memberspec:
      $brooklyn:entitySpec:
        type: org.apache.brooklyn.entity.webapp.jboss.JBoss7Server
        brooklyn.config:
          wars.root: http://search.maven.org/remotecontent?filepath=org/apache/brooklyn/example/brooklyn-example-hello-world-sql-webapp/0.8.0-incubating/brooklyn-example-hello-world-sql-webapp-0.8.0-incubating.war
          http.port: 8080+
          java.sysprops:
            brooklyn.example.db.url:
              $brooklyn:formatString:
                - jdbc:%s%s?user=%s\\&password=%s
                - $brooklyn:component("db").attributeWhenReady("datastore.url")
                - visitors
                - brooklyn
                - $brooklyn:external("brooklyn-demo-sample", "hidden-brooklyn-password")
- type: org.apache.brooklyn.entity.database.mysql.MySqlNode
  id: db
  name: DB HelloWorld Visitors
  brooklyn.config:
    creation.script.password: $brooklyn:external("brooklyn-demo-sample", "hidden-brooklyn-password")
    datastore.creation.script.url: https://github.com/apache/brooklyn-library/blob/master/examples/simple-web-cluster/src/main/resources/visitors-creation-script.sql
```

This sets up Nginx as the controller by default, but that can be configured using the `controllerspec` key. This uses the same `externalized config` as in other examples to hide the password.

JBoss is actually the default appserver in the `ControlledDynamicWebAppCluster`, so because `brooklyn.config` keys in Brooklyn are inherited by default, the same `blueprint` can be expressed more concisely as:

```
name: appserver-clustered-w-db-concise
services:
- type: org.apache.brooklyn.entity.webapp.ControlledDynamicWebAppCluster
  brooklyn.config:
    cluster.initial.size: 2
    wars.root: http://search.maven.org/remotecontent?filepath=org/apache/brooklyn/example/brooklyn-example-hello-world-sql-webapp/0.8.0-incubating/brooklyn-example-hello-world-sql-webapp-0.8.0-incubating.war
    http.port: 8080+
    java.sysprops:
      brooklyn.example.db.url: $brooklyn:formatString("jdbc:%s%s?user=%s\\&password=%s", component("db").attributeWhenReady("datastore.url"), "visitors", "brooklyn", $brooklyn:external("brooklyn-demo-sample", "hidden-brooklyn-password"))
- type: org.apache.brooklyn.entity.database.mysql.MySqlNode
  id: db
  name: DB HelloWorld Visitors
  brooklyn.config:
    creation.script.password: $brooklyn:external("brooklyn-demo-sample", "hidden-brooklyn-password")
    datastore.creation.script.url: https://github.com/apache/brooklyn-library/blob/master/examples/simple-web-cluster/src/main/resources/visitors-creation-script.sql
```

The other nicety supplied by the `ControlledDynamicWebAppCluster` blueprint is that it aggregates sensors from the appserver, so we have access to things like `webapp.reqs.perSec.windowed.perNode`. These are convenient for plugging in to policies! We can set up our blueprint to do autoscaling based on requests per second (keeping it in the range 10..100, with a maximum of 5 appserver nodes) as follows:

```
name: appserver-w-policy
services:
- type: org.apache.brooklyn.entity.webapp.ControlledDynamicWebAppCluster
  brooklyn.config:
    cluster.initial.size: 1
    dynamiccluster.memberspec:
      $brooklyn:entitySpec:
        type: org.apache.brooklyn.entity.webapp.jboss.JBoss7Server
        brooklyn.config:
          wars.root: http://search.maven.org/remotecontent?filepath=org/apache/brooklyn/example/brooklyn-example-hello-world-sql-webapp/0.8.0-incubating/brooklyn-example-hello-world-sql-webapp-0.8.0-incubating.war
          http.port: 8080+
          java.sysprops:
            brooklyn.example.db.url:
              $brooklyn:formatString:
                - jdbc:~{user=~{password=~{
                - $brooklyn:component("db").attributeWhenReady("datastore.url")
                - visitors
                - brooklyn
                - $brooklyn:external("brooklyn-demo-sample", "hidden-brooklyn-password")
          brooklyn.policies:
            - type: org.apache.brooklyn.policy.autoscaling.AutoScalerPolicy
              brooklyn.config:
                metric: $brooklyn:sensor("brooklyn.entity.webapp.DynamicWebAppCluster", "webapp.reqs.perSec.windowed.perNode")
                metricLowerBound: 10
                metricUpperBound: 100
                minPoolSize: 1
                maxPoolSize: 5
- type: org.apache.brooklyn.entity.database.mysql.MySqlNode
  id: db
  name: DB HelloWorld Visitors
  brooklyn.config:
    creation.script.password: $brooklyn:external("brooklyn-demo-sample", "hidden-brooklyn-password")
    datastore.creation.script.url: https://github.com/apache/brooklyn-library/raw/master/examples/simple-web-cluster/src/main/resources/visitors-creation-script.sql
```

Use your favorite load-generation tool (`jmeter` is one good example) to send a huge volume of requests against the server and see the policies kick in to resize it.

Java Entities

Java blueprints are powerful, but also rather more difficult to write than [YAML](#). Advanced Java skills are required.

The main uses of Java-based blueprints are:

- Integration with a service's API (e.g. for an on-line DNS service). This could take advantage of existing Java-based clients, or of Java's flexibility to chain together multiple calls.
- Complex management logic, for example when the best practices for adding/removing nodes from a cluster is fiddly and has many conditionals.
- Where the developer has a strong preference for Java. Anything that can be done in [YAML](#) can be done in the Java API. Once the [blueprint](#) is added to the catalog, the use of Java will be entirely hidden from users of that [blueprint](#).

The Apache Brooklyn community is striving to make [YAML](#)-based blueprints as simple as possible - if you come across a use-case that is hard to do in [YAML](#) then please let the community know.

- [Creating from a Maven Archetype](#)
- [Defining and Deploying](#)
- [Handling Bundle Dependencies](#)
- [Topology, Dependencies, and Management Policies](#)
- [Common Classes and Entities](#)
- [Feeds](#)
- [Writing an Entity](#)
- [Custom Entity Development](#)
- [Service State](#)
- [Entitlements](#)

Creating from a Maven Archetype

Maven Archetype

Brooklyn includes a maven archetype, which can be used to create the project structure for developing a new Java [entity](#), and generating the OSGi bundle for it.

Generating the Project

The archetype can be used interactively, by running:

```
$ mvn archetype:generate
```

The user will be prompted for the archetype to use (i.e. group "org.apache.brooklyn" and artifact "brooklyn-archetype-quickstart"), as well as options for the project to be created.

Alternatively, all options can be supplied at the command line. For example, if creating a project named "autobrick" for "com.acme":

```
$ BROOKLYN_VERSION=1.0.0-SNAPSHOT
$ mvn archetype:generate \
  -DarchetypeGroupId=org.apache.brooklyn \
  -DarchetypeArtifactId=brooklyn-archetype-quickstart \
  -DarchetypeVersion=${BROOKLYN_VERSION} \
  -DgroupId=com.acme \
  -DartifactId=autobrick \
  -Dversion=0.1.0-SNAPSHOT \
  -DpackageName=com.acme.autobrick \
  -DinteractiveMode=false
```

This will create a directory with the artifact name (e.g. "autobrick" in the example above). Note that if run from a directory containing a pom, it will also modify that pom to add this as a module!

The project will contain an example Java [entity](#). You can test this using the supplied unit tests, and also replace it with your own code.

The `README.md` file within the project gives further guidance.

Building

To build, run the commands:

```
$ cd autobrick
$ mvn clean install
```

Adding to the Catalog

The build will produce an OSGi bundle in `target/autobrick-0.1.0-SNAPSHOT.jar`, suitable for use in the [Brooklyn catalog](#) (using `brooklyn.libraries`).

To use this in your Brooklyn catalog you will first have to copy the target jar to a suitable [location](#). For developing/testing purposes storing on the local filesystem is fine. For production use, we recommend uploading to a remote maven repository or similar.

Once your jar is in a suitable [location](#) the next step is to add a new catalog item to Brooklyn. The project comes with a `catalog.bom` file, located in `src/main/resources`. Modify this file by adding a 'brooklyn.libraries' statement to the bom pointing to the jar. For example:

```
brooklyn.catalog:
  brooklyn.libraries:
    - file:///path/to/jar/autobrick-0.1.0-SNAPSHOT.jar
    version: "0.1.0-SNAPSHOT"
    itemType: entity
  items:
    - id: com.acme.autobrick.MySample
      item:
        type: com.acme.autobrick.MySample
```

The command below will use the CLI to add this to the catalog of a running Brooklyn instance:

```
br catalog add catalog.bom
```

After running that command, the OSGi bundle will have been added to the OSGi container, and the [entity](#) will have been added to your catalog. It can then be used in the same way as regular Brooklyn entities.

For example, you can use the [blueprint](#):

```
services:
  - type: com.acme.autobrick.MySample
```

Testing Entities

The project comes with unit tests that demonstrate how to test entities, both within Java and also using [YAML](#)-based blueprints.

A strongly recommended way is to write a [YAML](#) test [blueprint](#) using the test framework, and making this available to anyone who will use your [entity](#). This will allow users to easily run the test [blueprint](#) in their own environment (simply by deploying it to their own Brooklyn server) to confirm that the [entity](#) is working as expected. An example is contained within the project at `src/test/resources/sample-test.yaml`.

Defining and Deploying

Intro

This walkthrough will set up a simple **entity**, add it to the catalog, and provision it.

For illustration purposes, we will write an integration with [Github Gist](#), with an [effector](#) to create new gists.

Project Setup

Follow the instructions to create a new Java project using the `archetype`, and import it into your `favorite IDE`. This example assumes you used the groupId `com.acme` and artifact id `autobrick`.

First ensure you can build this project at the command line, using `mvn clean install`.

Java Entity Classes

For this particular example, we will use a third party Gist library, so will need to add that as a dependency. Add the following to your `pom.xml` inside the `<dependencies>` section (see [Maven](#) for more details):

```
<dependency>
  <groupId>org.eclipse.mylyn.github</groupId>
  <artifactId>org.eclipse.egit.github.core</artifactId>
  <version>2.1.5</version>
</dependency>
```

Create a new Java interface, `GistGenerator`, to describe the `entity`'s interface (i.e. the configuration options, sensors, and effectors). The code below assumes you have created this in the package `com.acme` for `src/main/java`.

```
package com.acme;

import java.io.IOException;

import org.apache.brooklyn.api.entity.Entity;
import org.apache.brooklyn.api.entity.ImplementedBy;
import org.apache.brooklyn.config.ConfigKey;
import org.apache.brooklyn.core.annotation.Effector;
import org.apache.brooklyn.core.annotation.EffectorParam;
import org.apache.brooklyn.core.config.ConfigKeys;

@ImplementedBy(GistGeneratorImpl.class)
public interface GistGenerator extends Entity {

    ConfigKey<String> OAUTH_KEY = ConfigKeys.newStringConfigKey("oauth.key", "OAuth key for creating a gist",
        "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx");

    @Effector(description="Create a Gist")
    String createGist(
        @EffectorParam(name="gistName", description="Gist Name", defaultValue="Demo Gist") String gistName,
        @EffectorParam(name="fileName", description="File Name", defaultValue="Hello.java") String fileName
    ,
        @EffectorParam(name="gistContents", description="Gist Contents", defaultValue="System.out.println(\n"
        + "Hello World\\");") String gistContents,
        @EffectorParam(name="oauth.key", description="OAuth key for creating a gist", defaultValue="") Strin
```

```

    ng oauthKey) throws IOException;

    @Effector(description="Retrieve a Gist")
    public String getGist(
        @EffectorParam(name="id", description="Gist id") String id,
        @EffectorParam(name="oauth.key", description="OAuth key for creating a gist", defaultValue="") String
    ng oauthKey) throws IOException;
}

```

To describe each part of this:

- The `@ImplementedBy` indicates the implementation class for this `entity` type - i.e. the class to instantiate when an `entity` of this type is created.
- By extending `Entity`, we indicate that this interface is an `Entity` type. We could alternatively have extended one of the other sub-types of `Entity`.
- The `oauth_key` is a configuration key - it is configuration that can be set on the `entity` when it is being instantiated.
- The `@Effector` annotation indicates that the given method is an `effector`, so should be presented and tracked as such. Execution of the `effector` is intercepted, to track it as a task and show its execution in the Activity view.
- The `@EffectorParam` annotations give metadata about the `effector`'s parameters. This metadata, such as the parameter description, is available to those using the client CLI, rest API and web-console.

Note there is an alternative way of defining effectors - adding them to the `entity` dynamically, discussed in the section [Dynamically Added Effectors](#).

Next lets add the implementation. Create a new Java class named `GistGeneratorImpl`.

```

package com.acme;

import java.io.IOException;
import java.util.Collections;

import org.apache.brooklyn.core.entity.AbstractEntity;
import org.apache.brooklyn.util.text.Strings;
import org.eclipse.egit.github.core.Gist;
import org.eclipse.egit.github.core.GistFile;
import org.eclipse.egit.github.core.service.GistService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.google.common.collect.Iterables;

public class GistGeneratorImpl extends AbstractEntity implements GistGenerator {

    private static final Logger LOG = LoggerFactory.getLogger(GistGeneratorImpl.class);

    @Override
    public String createGist(String gistName, String fileName, String gistContents, String oauthToken) throws IOException {
        if (Strings.isBlank(oauthToken)) oauthToken = config().get(OAUTH_KEY);

        GistFile file = new GistFile();
        file.setContent(gistContents);
        Gist gist = new Gist();
        gist.setDescription(gistName);
        gist.setFiles(Collections.singletonMap(fileName, file));
        gist.setPublic(true);

        GistService service = new GistService();
        service.getClient().setOAuth2Token(oauthToken);
        LOG.info("Creating Gist: " + gistName);
        Gist result = service.createGist(gist);
        return result.getId();
    }
}

```



```

@Override
public String getGist(String id, String oathToken) throws IOException {
    if (Strings.isBlank(oathToken)) oathToken = config().get(OAUTH_KEY);

    GistService service = new GistService();
    service.getClient().setOAuth2Token(oathToken);
    Gist gist = service.getGist(id);
    return Iterables.getOnlyElement(gist.getFiles().values()).getContent();
}
}

```

To describe each part of this:

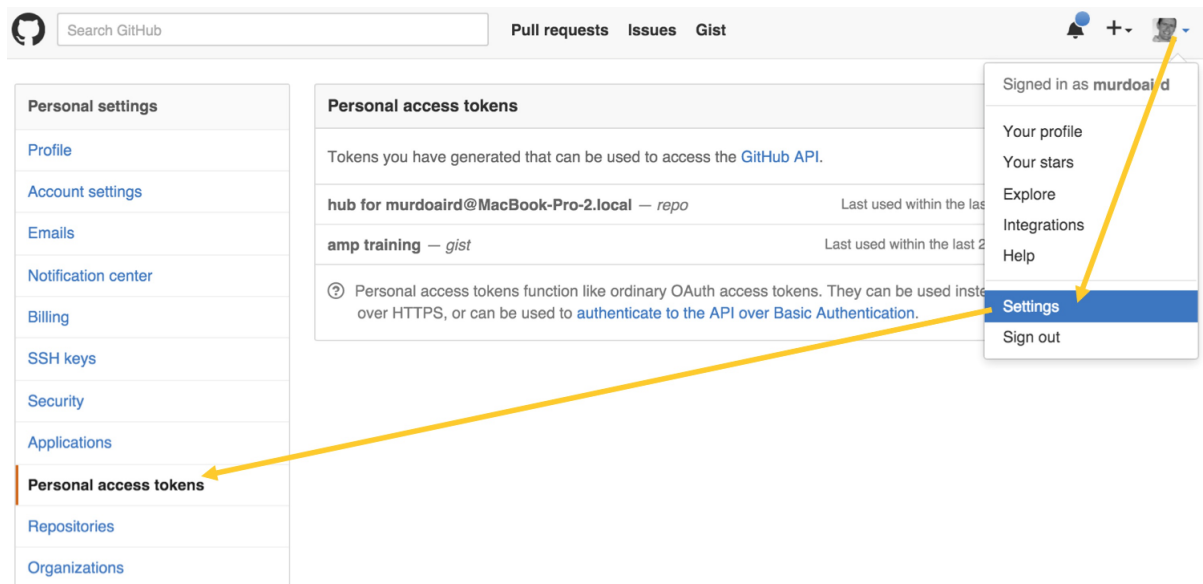
- Extends `AbstractEntity` - all `entity` implementations should extend this, or one of its sub-types.
- Implements `gistGenerator` : this is the `Entity` type definition, so must be implemented. Users of the `entity` will only refer to the interface; they will never be given an instance of the concrete class - instead a `dynamic proxy` is used (to allow remoting).
- `org.slf4j.Logger` is the logger used throughout Apache Brooklyn.
- Implements the `createGist` `effector` - we do not need to re-declare all the annotations.
- If no `oath.key` parameter was passed in, then use the configuration set on the `entity`.
- Use the third party library to create the gist.

Configuring GitHub

First, create a github.com account, if you do not already have one.

Before running the `blueprint`, we'll need to generate an access token that has permissions to create a gist programmatically.

First `create a new access token` that our `blueprint` will use to create a gist:



Next, grant the token rights to create gists:

Token description

amp training

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes.](#)

<input type="checkbox"/> repo	Full control of private repositories
<input type="checkbox"/> repo:status	Access commit status
<input type="checkbox"/> repo_deployment	Access deployment status
<input type="checkbox"/> public_repo	Access public repositories
<input type="checkbox"/> admin:org	Full control of orgs and teams
<input type="checkbox"/> write:org	Read and write org and team membership
<input type="checkbox"/> read:org	Read org and team membership
<input type="checkbox"/> admin:public_key	Full control of user public keys
<input type="checkbox"/> write:public_key	Write user public keys
<input type="checkbox"/> read:public_key	Read user public keys
<input type="checkbox"/> admin:repo_hook	Full control of repository hooks
<input type="checkbox"/> write:repo_hook	Write repository hooks
<input type="checkbox"/> read:repo_hook	Read repository hooks
<input type="checkbox"/> admin:org_hook	Full control of organization hooks
<input checked="" type="checkbox"/> gist	Create gists
<input type="checkbox"/> notifications	Access notifications
<input type="checkbox"/> user	Update all user data
<input type="checkbox"/> user:email	Access user email addresses (read-only)
<input type="checkbox"/> user:follow	Follow and unfollow users
<input type="checkbox"/> delete_repo	Delete repositories

Grant the token rights to create gists

Testing

The archetype project comes with example unit tests that demonstrate how to test entities, both within Java and also using **YAML**-based blueprints.

We will create a similar Java-based test for this [blueprint](#). Create a new Java class named `GistGeneratorTest` in the package `com.acme`, inside `src/test/java`.

You will need to substitute the github access token you generated in the previous section for the placeholder text

XX .

```
package com.acme;

import static org.testng.Assert.assertEquals;

import org.apache.brooklyn.api.entity.EntitySpec;
import org.apache.brooklyn.core.test.BrooklynAppUnitTestSupport;
import org.testng.annotations.Test;

public class GistGeneratorTest extends BrooklynAppUnitTestSupport {

    @Test
    public void testEntity() throws Exception {
        String oathKey = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
        GistGenerator entity = app.createAndManageChild(EntitySpec.create(GistGenerator.class));
        String id = entity.createGist("myGistName", "myFileName", "myGistContents", oathKey);

        String contents = entity.getGist(id, oathKey);
        assertEquals(contents, "myGistContents");
    }
}
```

Similarly, we can write a test that uses the `GistGenerator` from a `YAML blueprint`. Create a new Java class named `GistGeneratorYamlTest` in the package `com.acme`, inside `src/test/java`.

Again you will need to substitute the github access token you generated in the previous section for the placeholder text xx . See the section on externalised configuration for how to store these credentials more securely.

```
package com.acme;

import static org.testng.Assert.assertEquals;

import org.apache.brooklyn.api.entity.Entity;
import org.apache.brooklyn.camp.brooklyn.AbstractYamlTest;
import org.apache.brooklyn.core.entity.Entities;
import org.testng.annotations.Test;

import com.google.common.base.Joiner;
import com.google.common.collect.Iterables;

public class GistGeneratorYamlTest extends AbstractYamlTest {

    private String contents;

    @Test
    public void testEntity() throws Exception {
        String oathKey = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";

        String yaml = Joiner.on("\n").join(
            "name: my test",
            "services:",
            "- type: com.acme.GistGenerator",
            "  brooklyn.config:",
            "    oauth.key: "+oathKey);

        Entity app = createAndStartApplication(yaml);
        waitForApplicationTasks(app);

        Entities.dumpInfo(app);

        GistGenerator entity = (GistGenerator) Iterables.getOnlyElement(app.getChildren());
        String id = entity.createGist("myGistName", "myFileName", "myGistContents", null);

        contents = entity.getGist(id, null);
        assertEquals(contents, "myGistContents");
    }
}
```

Building the OSGi Bundle

Next we will build this example as an **OSGi Bundle** so that it can be added to the Apache Brooklyn server at runtime, and so multiple versions of the **blueprint** can be managed.

The `mvn clean install` will automatically do this, creating a jar inside the `target/` sub-directory of the project. This works by using the [Maven Bundle Plugin](#) which we get automatically by declaring the `pom.xml`'s parent as `brooklyn-downstream-parent`.

Adding to the catalog

Similar to the `sample.bom` entity that ships with the archetype, we will define a `.bom` file to add our `gistGenerator` to the catalog. Substitute the URL below for your own newly built artifact (which will be in the `target` sub-directory after running `mvn clean install`).

The command below will use the `br` CLI to add this to the catalog of a running Brooklyn instance. Substitute the credentials, URL and port for those of your server.

Handling Bundle Dependencies

Some Java blueprints will require third party libraries. These need to be made available to the Apache Brooklyn runtime. There are a number of ways this can be achieved.

Classic Mode: Dropins Folder

In Brooklyn classic mode (i.e. when not using Karaf), jars can be added to `./lib/dropins/`. After restarting Brooklyn, these will be available on the classpath.

In Brooklyn classic mode, there is an embedded OSGi container. This is used for installing libraries referenced in catalog items.

OSGi Bundles

Introduction to OSGi Bundles

An [OSGi bundle](#) is a jar file with additional metadata in its manifest file. The `MANIFEST.MF` file contains the symbolic name and version of the bundle, along with details of its dependencies and of the packages it exports (which are thus visible to other bundles).

The [maven-bundle-plugin](#) is a convenient way of building OSGi bundles.

OSGi Bundles Declared in Catalog Items

Within a [catalog item](#), a list of URLs can be supplied under `brooklyn.libraries`. Each URL should point to an OSGi bundle. This list should include the OSGi bundle that has the Java code for your [blueprint](#), and also the OSGi bundles that it depends on (including all transitive dependencies).

It is vital that these jars are built correctly as OSGi bundles, and that all transitive dependencies are included. The bundles will be added to Karaf in the order given, so a bundle's dependencies should be listed before the bundle(s) that depend on them.

In the [GistGenerator example](#), the `catalog.bom` file included the URL of the dependency `org.eclipse.egit.github.core`. It also (before that line) included its transitive dependency, which is a specific version of `gson`.

For Java [blueprint](#) developers, this is often the most convenient way to share a [blueprint](#).

Similarly for those wishing to use a new [blueprint](#), this is often the simplest mechanism: the dependencies are fully described in the catalog item, which makes it convenient for deploying to Apache Brooklyn instances where there is not direct access to Karaf or the file system.

Adding Bundles and Features Directly to Karaf

Bundles and features can be added manually, directly to Karaf.

However, note this only affects the single Karaf instance. If running in HA mode or if provisioning a new instance of Apache Brooklyn, the bundles will also need to be added to these Karaf instances.

Karaf Console

Login to the [Karaf console](#) using `./bin/client`, and add the bundles and features as desired.

Examples of some useful commands are shown below:

```
karaf@amp> bundle:install -s http://repo1.maven.org/maven2/org/apache/servicemix/bundles/org.apache.servicemix.bundles.egit.github.core/2.1.5.1/org.apache.servicemix.bundles.egit.github.core-2.1.5.1.jar
Bundle ID: 316

karaf@amp> bundle:list -t 0 -s | grep github
318 | Active   | 80 | 2.1.5.1                | org.apache.servicemix.bundles.egit.github.core

karaf@amp> bundle:headers org.apache.servicemix.bundles.egit.github.core
...

karaf@amp> bundle:uninstall org.apache.servicemix.bundles.egit.github.core
```

Karaf Deploy Folder

Karaf support [hot deployment](#). There are a set of deployers, such as feature and KAR deployers, that handle deployment of artifacts added to the `deploy` folder.

Note that the Karaf console can give finer control (including for uninstall).

Karaf KAR files

[Karaf KAR](#) is an archive format (Karaf ARchive). A KAR is a jar file (so a zip file), which contains a set of feature descriptors and bundle jar files.

This can be a useful way to bundle a more complex Java [blueprint](#) (along with its dependencies), to make it easier for others to install.

A KAR file can be built using the [maven plugin org.apache.karaf.tooling:features-maven-plugin](#).

Karaf Features

A [karaf feature.xml](#) defines a set of bundles that make up a feature. Once a feature is defined, one can add it to a Karaf instance: either directly (e.g. using the [Karaf console](#)), or by referencing it in another feature.xml file.

Embedded Dependencies

An OSGi bundle can [embed jar dependencies](#) within it. This allows dependencies to be kept private within a bundle, and easily shipped with that bundle.

To keep these private, it is vital that the OSGi bundle does not import or export the packages contained within those embedded jars, and does not rely on any of those packages in the public signatures of any packages that are exported or imported.

Converting Non-OSGi Dependencies to Bundles

If a dependencies is not available as an OSGi bundle (and you don't want to just [embed the jar](#)), there are a few options for getting an equivalent OSGi bundle:

- Use a ServiceMix re-packaged jar, if available. ServiceMix have re-packed many common dependencies as OSGi bundles, and published them on [Maven Central](#).

- Use the `wrap:` prefix. The [PAX URL Wrap protocol](#) is an OSGi URL handler that can process your legacy jar at runtime and transform it into an OSGi bundle. This can be used when declaring a dependency in your feature.xml, and when using the Karaf console's `bundle:install` . Note that it is not yet supported in Brooklyn's `brooklyn.libraries` catalog items.
- Re-package the bundle yourself, offline, to produce a valid OSGi bundle.

Topology, Dependencies, and Management Policies

Applications written in [YAML](#) can similarly be written in Java. However, the [YAML](#) approach is recommended.

Define your Application Blueprint

The example below creates a three tier web service, composed of an Nginx load-balancer, a cluster of Tomcat app-servers, and a MySQL database. It is similar to the [YAML policies example](#), but also includes the MySQL database to demonstrate the use of dependent configuration.

```
package com.acme.autobrick;

import org.apache.brooklyn.api.entity.EntitySpec;
import org.apache.brooklyn.api.policy.PolicySpec;
import org.apache.brooklyn.api.sensor.AttributeSensor;
import org.apache.brooklyn.api.sensor.EnricherSpec;
import org.apache.brooklyn.core.entity.AbstractApplication;
import org.apache.brooklyn.core.sensor.DependentConfiguration;
import org.apache.brooklyn.core.sensor.Sensors;
import org.apache.brooklyn.enricher.stock.Enrichers;
import org.apache.brooklyn.entity.database.mysql.MySqlNode;
import org.apache.brooklyn.entity.group.DynamicCluster;
import org.apache.brooklyn.entity.proxy.nginx.NginxController;
import org.apache.brooklyn.entity.webapp.tomcat.TomcatServer;
import org.apache.brooklyn.policy.autoscaling.AutoScalerPolicy;
import org.apache.brooklyn.policy.ha.ServiceFailureDetector;
import org.apache.brooklyn.policy.ha.ServiceReplacer;
import org.apache.brooklyn.policy.ha.ServiceRestarter;
import org.apache.brooklyn.util.time.Duration;

public class ExampleWebApp extends AbstractApplication {

    @Override
    public void init() {
        AttributeSensor<Double> reqsPerSecPerNodeSensor = Sensors.newDoubleSensor(
            "webapp.reqs.perSec.perNode",
            "Reqs/sec averaged over all nodes");

        MySqlNode db = addChild(EntitySpec.create(MySqlNode.class)
            .configure(MySqlNode.CREATION_SCRIPT_URL, "https://bit.ly/brooklyn-visitors-creation-script"));

        DynamicCluster cluster = addChild(EntitySpec.create(DynamicCluster.class)
            .displayName("Cluster")
            .configure(DynamicCluster.MEMBER_SPEC, EntitySpec.create(TomcatServer.class)
                .configure(TomcatServer.ROOT_WAR,
                    "http://search.maven.org/remotecontent?filepath=org/apache/brooklyn/example/brooklyn-example-hello-world-sql-webapp/0.8.0-incubating/brooklyn-example-hello-world-sql-webapp-0.8.0-incubating-war")
                .configure(TomcatServer.JAVA_SYSPROPS.subKey("brooklyn.example.db.url"),
                    DependentConfiguration.formatString("jdbc:%s%s?user=%s&password=%s",
                        DependentConfiguration.attributeWhenReady(db, MySqlNode.DATASTORE_URL),
                        "visitors", "brooklyn", "br00k11n")))
            .policy(PolicySpec.create(ServiceRestarter.class)
                .configure(ServiceRestarter.FAIL_ON_RECURRING_FAILURES_IN_THIS_DURATION, Duration.seconds(5)))
            .enricher(EnricherSpec.create(ServiceFailureDetector.class)
                .configure(ServiceFailureDetector.ENTITY_FAILED_STABILIZATION_DELAY, Duration.seconds(30))))
            .policy(PolicySpec.create(ServiceReplacer.class))
            .policy(PolicySpec.create(AutoScalerPolicy.class)
                .configure(AutoScalerPolicy.METRIC, reqsPerSecPerNodeSensor))
    }
}
```



```

        .configure(AutoScalerPolicy.METRIC_LOWER_BOUND, 1)
        .configure(AutoScalerPolicy.METRIC_UPPER_BOUND, 3)
        .configure(AutoScalerPolicy.RESIZE_UP_STABILIZATION_DELAY, Duration.seconds(2))
        .configure(AutoScalerPolicy.RESIZE_DOWN_STABILIZATION_DELAY, Duration.minutes(1))
        .configure(AutoScalerPolicy.MAX_POOL_SIZE, 3))
    .enricher(Enrichers.builder().aggregating(TomcatServer.REQUESTS_PER_SECOND_IN_WINDOW)
        .computingAverage()
        .fromMembers()
        .publishing(reqsPerSecPerNodeSensor)
        .build());
    addChild(EntitySpec.create(NginxController.class)
        .configure(NginxController.SERVER_POOL, cluster)
        .configure(NginxController.STICKY, false));
}
}

```

To describe each part of this:

- The application extends `AbstractApplication`.
- It implements `init()`, to add its child entities. The `init` method is called only once, when instantiating the `entity` instance.
- The `addChild` method takes an `EntitySpec`. This describes the `entity` to be created, defining its type and its configuration.
- The `brooklyn.example.db.url` is a system property that will be passed to each `TomcatServer` instance. Its value is the database's URL (discussed below).
- The policies and enrichers provide in-life management of the application, to restart failed instances and to replace those components that repeatedly fail.
- The `NginxController` is the load-balancer and reverse-proxy: by default, it round-robins to the ip:port of each member of the cluster configured as the `SERVER_POOL`.

Dependent Configuration

Often a component of an application will depend on another component, where the dependency information is only available at runtime (e.g. it requires the IP of a dynamically provisioned component). For example, the app-servers in the example above require the database URL to be injected.

The "DependentConfiguration" methods returns a future (or a "promise" in the language of some other programming languages): when the value is needed, the caller will block to wait for the future to resolve. It will block only "at the last moment" when the value is needed (e.g. after the VMs have been provisioned and the software is installed, thus optimising the provisioning time). It will automatically monitor the given `entity's sensor`, and generate the value when the `sensor` is populated.

The `attributeWhenReady` is used to generate a configuration value that depends on the dynamic `sensor` value of another `entity` - in the example above, it will not be available until that `MySQLNode.DATABASE_URL` `sensor` is populated. At that point, the JDBC URL will be constructed (as defined in the `formatString` method, which also returns a future).

Common Classes and Entities

Entity Class Hierarchy

By convention in Brooklyn the following words have a particular meaning:

- *Group* - a homogeneous grouping of entities (which need not all be managed by the same parent *entity*)
- *Cluster* - a homogeneous collection of entities (all managed by the "cluster" *entity*)
- *Fabric* - a multi-*location* collection of entities, with one per *location*; often used with a cluster per *location*
- *Application* - a top-level *entity*, which can have one or more child entities.

The following constructs are often used for Java entities:

- *entity spec* defines an *entity* to be created; used to define a child *entity*, or often to define the type of *entity* in a cluster.
- *traits* (mixins) providing certain capabilities, such as *Resizable* and *Startable*.
- *Resizable* entities can re-sized dynamically, to increase/decrease the number of child entities. For example, scaling up or down a cluster. It could similarly be used to vertically scale a VM, or to resize a disk.
- *Startable* indicates the *effector* to be executed on initial deployment (*start()*) and on tear down (*stop()*).

Configuration

Configuration keys are typically defined as static named fields on the *Entity* interface. These define the configuration values that can be passed to the *entity* during construction. For example:

```
public static final ConfigKey<String> ROOT_WAR = new ConfigKeys.newStringConfigKey(
    "wars.root",
    "WAR file to deploy as the ROOT, as URL (supporting file: and classpath: prefixes)");
```

If supplying a default value, it is important that this be immutable. Otherwise, it risks users of the *blueprint* modifying the default value, which would affect blueprints that are subsequently deployed.

One can optionally define a `@SetFromFlag("war")` . This defines a short-hand for configuring the *entity*. However, it should be used with caution - when using configuration set on a parent *entity* (and thus inherited), the `@SetFromFlag` short-form names are not checked. The long form defined in the constructor should be meaningful and sufficient. The usage of `@SetFromFlag` is therefore discouraged.

The type `AttributeSensorAndConfigKey<?>` can be used to indicate that a config key should be resolved, and its value set as a *sensor* on the *entity* (when `configToAttributes.apply(entity)` is called).

A special case of this is `PortAttributeSensorAndConfigKey` . This is resolved to find an available port (by querying the target *location*). For example, the value `8081+` means that then next available port starting from 8081 will be used.

Declaring Sensors

Sensors are typically defined as static named fields on the *Entity* interface. These define the events published by the *entity*, which interested parties can subscribe to. For example:

```
AttributeSensor<String> MANAGEMENT_URL = Sensors.newStringSensor(
    "crate.managementUri",
    "The address at which the Crate server listens");
```

Declaring Effectors

Effectors are the operations that an **entity** supports. There are multiple ways that an **entity** can be defined. Examples of each are given below.

Effector Annotation

A method on the **entity** interface can be annotated to indicate it is an **effector**, and to provide metadata about the **effector** and its parameters.

```
@org.apache.brooklyn.core.annotation.Effector(description="Retrieve a Gist")
public String getGist(@EffectorParam(name="id", description="Gist id") String id);
```

Static Field Effector Declaration

A static field can be defined on the **entity** to define an **effector**, giving metadata about that **effector**.

```
public static final Effector<String> EXECUTE_SCRIPT = Effectors.effector(String.class, "executeScript")
    .description("invokes a script")
    .parameter(ExecuteScriptEffectorBody.SCRIPT)
    .impl(new ExecuteScriptEffectorBody())
    .build();
```

In this example, the implementation of the **effector** is an instance of `ExecuteScriptEffectorBody`. This implements `EffectorBody`. It will be invoked whenever the **effector** is called.

Dynamically Added Effectors

An **effector** can be added to an **entity** dynamically - either as part of the **entity**'s `init()` or as separate initialization code. This allows the implementation of the **effector** to be shared amongst multiple entities, without sub-classing. For example:

```
Effector<Void> GET_GIST = Effectors.effector(Void.class, "createGist")
    .description("Create a Gist")
    .parameter(String.class, "id", "Gist id")
    .buildAbstract();

public static void CreateGistEffectorBody implements EffectorBody<Void>() {
    @Override
    public Void call(ConfigBag parameters) {
        // impl
        return null;
    }
}

@Override
public void init() {
    getMutableEntityType().addEffector(CREATE_GIST, new CreateGistEffectorBody());
}
```

Effector Invocation

There are several ways to invoke an **effector** programmatically:

- Where there is an annotated method, simply call the method on the interface.

- Call the `invoke` method on the `entity`, using the static `effector` declaration. For example:

```
entity.invoke(CREATE_GIST, ImmutableMap.of("id", id)); .
```
- Call the utility method `org.apache.brooklyn.core.entity.Entities.invokeEffector` . For example:

```
Entities.invokeEffector(this, targetEntity, CREATE_GIST, ImmutableMap.of("id", id)); .
```

When an `effector` is invoked, the call is intercepted to wrap it in a task. In this way, the `effector` invocation is tracked - it is shown in the Activity view.

When `invoke` or `invokeEffector` is used, the call returns a `Task` object (which extends `Future`). This allows the caller to understand progress and errors on the task, as well as calling `task.get()` to retrieve the return value. Be aware that `task.get()` is a blocking function that will wait until a value is available before returning.

Tasks

Warning: the task API may be changed in a future release. However, backwards compatibility will be maintained where possible.

When implementing entities and policies, all work done within Brooklyn is executed as Tasks. This makes it trackable and visible to administrators. For the activity list to show a break-down of an `effector`'s work (in real-time, and also after completion), tasks and sub-tasks must be created.

In common situations, tasks are implicitly created and executed. For example, when implementing an `effector` using the `@Effector` annotation on a method, the method invocation is automatically wrapped as a task. Similarly, when a subscription is passed an event (e.g. when using `SensorEventListener.onEvent(SensorEvent<T> event)`), that call is done inside a task.

Within a task, it is possible to create and execute sub-tasks. A common way to do this is to use `dynamicTasks.queue` . If called from within a "task queuing context" (e.g. from inside an `effector` implementation), it will add the task to be executed. By default, the outer task will not be marked as done until its queued sub-tasks are complete.

When creating tasks, the `TaskBuilder` can be used to create simple tasks or to create compound tasks whose sub-tasks are to be executed either sequentially or in parallel. For example:

```
TaskBuilder.<Integer>builder()
    .displayName("stdout-example")
    .body(new Callable<Integer>() { public Integer call() { System.out.println("example"); } })
    .build();
```

There are also builder and factory utilities for common types of operation, such as executing SSH commands using `SshTasks` .

A lower level way to submit tasks within an `entity` is to call `getExecutionContext().submit(...)` . This automatically tags the task to indicate that its context is the given `entity`.

An even lower level way to execute tasks (to be ignored except for power-users) is to go straight to the `getManagementContext().getExecutionManager().submit(...)` . This is similar to the standard Java `Executor` , but also supports more metadata about tasks such as descriptions and tags. It also supports querying for tasks. There is also support for submitting `ScheduledTask` instances which run periodically.

The `Tasks` and `BrooklynTaskTags` classes supply a number of conveniences including builders to make working with tasks easier.

Subscriptions and the Subscription Manager

Entities, locations, policies and enrichers can subscribe to events. These events could be attribute-change events from other entities, or other events explicitly published by the entities.

A subscription is created by calling `subscriptions().subscribe(entity, sensorType, sensorEventListener)`. The `sensorEventListener` will be called with the event whenever the given `entity` emits a `sensor` of the given type. If `null` is used for either the `entity` or `sensor` type, this is treated as a wildcard.

It is very common for a `policy` or `enricher` to subscribe to events, to kick off actions or to publish other aggregated attributes or events.

Feeds

Feeds

`Feed`s within Apache Brooklyn are used to populate an `entity`'s sensors. There are a variety of feed types, which commonly poll to retrieve the raw metrics of the `entity` (for example polling an HTTP management API, or over JMX).

Persistence

There are two ways to associate a feed with an `entity`.

The first way is (within the `entity`) to call `feeds().addFeed(...)`. This persists the feed: the feed will be automatically added to the `entity` when the Brooklyn server restarts. It is important that all configuration of the feed is persistable (e.g. not using any in-line anonymous inner classes to define functions).

The feed builders can be passed a `uniqueTag(...)`, which will be used to ensure that on rebind there will not be multiple copied of the feed (e.g. if `rebind()` had already re-created the feed).

The second way is to just pass to the feed's builder the `entity`. When using this mechanism, the feed will be wired up to the `entity` but it will not be persisted. In this case, it is important that the `entity`'s `rebind()` method recreates the feed.

Types of Feed

HTTP Feed

An `HttpFeed` polls over http(s). An example is shown below:

```
private HttpFeed feed;

@Override
protected void connectSensors() {
    super.connectSensors();

    feed = feeds().addFeed(HttpFeed.builder()
        .period(200)
        .baseUri(String.format("http://%s:%s/management/subsystem/web/connector/http/read-resource", host, port))
        .baseUriVars(ImmutableMap.of("include-runtime", "true"))
        .poll(new HttpPollConfig(SERVICE_UP)
            .onSuccess(HttpValueFunctions.responseCodeEquals(200))
            .onError(Functions.constant(false)))
        .poll(new HttpPollConfig(REQUEST_COUNT)
            .onSuccess(HttpValueFunctions.jsonContents("requestCount", Integer.class)))
        .build());
}

@Override
protected void disconnectSensors() {
    super.disconnectSensors();
    if (feed != null) feed.stop();
}
```

SSH Feed

An SSH feed executes a command over ssh periodically. An example is shown below:

```
private AbstractCommandFeed feed;
```

```

@Override
protected void connectSensors() {
    super.connectSensors();

    feed = feeds.addFeed(SshFeed.builder()
        .machine(mySshMachineLachine)
        .poll(new CommandPollConfig(SERVICE_UP)
            .command("rabbitmqctl -q status")
            .onSuccess(new Function() {
                public Boolean apply(SshPollValue input) {
                    return (input.getExitStatus() == 0);
                }
            })))
        .build());
}

@Override
protected void disconnectSensors() {
    super.disconnectSensors();
    if (feed != null) feed.stop();
}

```

WinRm CMD Feed

A WinRM feed executes a windows command over winrm periodically. An example is shown below:

```

private AbstractCommandFeed feed;

//@Override
protected void connectSensors() {
    super.connectSensors();

    feed = feeds.addFeed(CmdFeed.builder()
        .entity(entity)
        .machine(machine)
        .poll(new CommandPollConfig<String>(SENSOR_STRING)
            .command("ipconfig")
            .onSuccess(SshValueFunctions.stdout()))
        .build());
}

@Override
protected void disconnectSensors() {
    super.disconnectSensors();
    if (feed != null) feed.stop();
}

```

Windows Performance Counter Feed

This type of feed retrieves performance counters from a Windows host, and posts the values to sensors.

One must supply a collection of mappings between Windows performance counter names and Brooklyn attribute sensors.

This feed uses WinRM to invoke the windows utility `typeperf` to query for a specific set of performance counters, by name. The values are extracted from the response, and published to the `entity`'s sensors. An example is shown below:

```

private WindowsPerformanceCounterFeed feed;

@Override
protected void connectSensors() {
    feed = feeds.addFeed(WindowsPerformanceCounterFeed.builder()
        .addSensor("\\Processor(_total)\\% Idle Time", CPU_IDLE_TIME)
        .addSensor("\\Memory\\Available MBytes", AVAILABLE_MEMORY)
        .build());
}

```

```

}

@Override
protected void disconnectSensors() {
    super.disconnectSensors();
    if (feed != null) feed.stop();
}

```

JMX Feed

This type of feed queries over JMX to retrieve **sensor** values. This can query attribute values or call operations.

The JMX connection details can be automatically inferred from the **entity**'s standard attributes, or it can be explicitly supplied.

An example is shown below:

```

private JmxFeed feed;

@Override
protected void connectSensors() {
    super.connectSensors();

    feed = feeds().addFeed(JmxFeed.builder()
        .period(5, TimeUnit.SECONDS)
        .pollAttribute(new JmxAttributePollConfig<Integer>(ERROR_COUNT)
            .objectName(requestProcessorMbeanName)
            .attributeName("errorCount"))
        .pollAttribute(new JmxAttributePollConfig<Boolean>(SERVICE_UP)
            .objectName(serverMbeanName)
            .attributeName("Started")
            .onError(Functions.constant(false)))
        .build());
}

@Override
protected void disconnectSensors() {
    super.disconnectSensors();
    if (feed != null) feed.stop();
}

```

Function Feed

This type of feed periodically executes something to compute the attribute values. This can be a **callable**, **Supplier** or Groovy **closure**. It must be persistable (e.g. not use an in-line anonymous inner classes).

An example is shown below:

```

public static class ErrorCountRetriever implements Callable<Integer> {
    private final Entity entity;

    public ErrorCountRetriever(Entity entity) {
        this.entity = entity;
    }

    @Override
    public Integer call() throws Exception {
        // TODO your implementation...
        return 0;
    }
}

private FunctionFeed feed;

@Override

```



```
protected void connectSensors() {
    super.connectSensors();

    feed = feeds().addFeed(FunctionFeed.builder()
        .poll(new FunctionPollConfig<Object, Integer>(ERROR_COUNT)
            .period(500, TimeUnit.MILLISECONDS)
            .callable(new ErrorCountRetriever(this))
            .onExceptionOrFailure(Functions.<Integer>constant(null))
        .build());
}

@Override
protected void disconnectSensors() {
    super.disconnectSensors();
    if (feed != null) feed.stop();
}
```

Writing an Entity

Ways to write an entity

There are several ways to write a new `entity`:

- For Unix/Linux, write `YAML` blueprints, for example using a `VanillaSoftwareProcess` and configuring it with your scripts.
- For Windows, write `YAML` blueprints using `VanillaWindowsProcess` and configure the PowerShell scripts.
- For composite entities, use `YAML` to compose exiting types of entities (potentially overwriting parts of their configuration), and wire them together.
- Use **Chef recipes**.
- Use **Salt formulas**.
- Use **Ansible playbooks**.
- Write pure-java, extending existing base-classes. For example, the `GistGenerator` [example](#). These can use utilities such as `HttpTool` and `BashCommands`.
- Write pure-Java blueprints that extend `SoftwareProcess`. However, the `YAML` approach is strongly recommended over this approach.
- Write pure-Java blueprints that compose together existing entities, for example to manage a cluster. Often this is possible in `YAML` and that approach is strongly recommended. However, sometimes the management logic may be so complex that it is easier to use Java.

The rest of this section covers writing an `entity` in pure-java (or other JVM languages).

Things To Know

All entities have an interface and an implementation. The methods on the interface are its effectors; the interface also defines its sensors.

Entities are created through the management context (rather than calling the constructor directly). This returns a proxy for the `entity` rather than the real instance, which is important in a distributed management plane.

All `entity` implementations inherit from `AbstractEntity`, often through one of the following:

- `SoftwareProcessImpl` : if it's a software process
- `VanillaJavaAppImpl` : if it's a plain-old-java app
- `JavaWebAppSoftwareProcessImpl` : if it's a JVM-based web-app
- `DynamicClusterImpl`, `DynamicGroupImpl` or `AbstractGroupImpl` : if it's a collection of other entities

Software-based processes tend to use *drivers* to install and launch the remote processes onto *locations* which support that driver type. For example, `AbstractSoftwareProcessSshDriver` is a common driver superclass, targetting `SshMachineLocation` (a machine to which Brooklyn can ssh). The various `softwareProcess` entities above (and some of the exemplars listed at the end of this page) have their own dedicated drivers.

Finally, there are a collection of *traits*, such as `Resizable`, in the package `brooklyn.entity.trait`. These provide common sensors and effectors on entities, supplied as interfaces. Choose one (or more) as appropriate.

Key Steps

NOTE: Consider instead writing a [YAML blueprint](#) for your [entity](#).

So to get started:

1. Create your [entity](#) interface, extending the appropriate selection from above, to define the effectors and sensors.
2. Include an annotation like `@ImplementedBy(YourEntityImpl.class)` on your interface, where `YourEntityImpl` will be the class name for your [entity](#) implementation.
3. Create your [entity](#) class, implementing your [entity](#) interface and extending the classes for your chosen [entity](#) super-types. Naming convention is a suffix "Impl" for the [entity](#) class, but this is not essential.
4. Create a driver interface, again extending as appropriate (e.g. `SoftwareProcessDriver`). The naming convention is to have a suffix "Driver".
5. Create the driver class, implementing your driver interface, and again extending as appropriate. Naming convention is to have a suffix "SshDriver" for an ssh-based implementation. The correct driver implementation is found using this naming convention, or via custom namings provided by the `BasicEntityDriverFactory`.
6. Wire the `public class getDriverInterface()` method in the [entity](#) implementation, to specify your driver interface.
7. Provide the implementation of missing lifecycle methods in your driver class (details below)
8. Connect the sensors from your [entity](#) (e.g. overriding `connectSensors()` of `SoftwareProcessImpl`). See the [sensor](#) feeds, such as `HttpFeed` and `JmxFeed`.

Any JVM language can be used to write an [entity](#). However use of pure Java is encouraged for entities in core brooklyn.

Helpful References

A few handy pointers will help make it easy to build your own entities. Check out some of the exemplar existing entities (note, some of the other entities use deprecated utilities and a deprecated class hierarchy; it is suggested to avoid these, looking at the ones below instead):

- `JBoss7Server`
- `MySQLNode`

You might also find the following helpful:

- [Entity Design Tips](#)
- The [User Guide](#)
- The [Mailing List](#)

Custom Entity Development

This section details how to create new custom application components or groups as brooklyn entities.

The Entity Lifecycle

- Importance of serialization, ref to How mananagement works
- Parents and Membership (groups)

What to Extend -- Implementation Classes

- `entity` implementation class hierarchy
 - `SoftwareProcess` as the main starting point for base entities (corresponding to software processes), and subclasses such as `VanillaJavaApp`
 - `DynamicCluster` (multiple instances of the same `entity` in a `location`) and `DynamicFabric` (clusters in multiple `location`) for automatically creating many instances, supplied with an `EntityFactory` (e.g. `BaseEntityFactory`) in the `factory` flag
 - `AbstractGroup` for collecting entities which are parented elsewhere in the hierarchy
 - `AbstractEntity` if nothing else fits
- traits (mixins, otherwise known as interfaces with statics) to define available config keys, sensors, and effectors; and conveniences e.g. `StartableMethods.{start,stop}` is useful for entities which implement `Startable`
- the `Entities` class provides some generic convenience methods; worth looking at it for any work you do

A common lifecycle pattern is that the `start` `effector` (see more on effectors below) is invoked, often delegating either to a driver (for software processes) or children entities (for clusters etc).

Configuration

- `AttributeSensorAndConfigKey` fields can be automatically converted for `SoftwareProcess`. This is done in `preStart()`. This must be done manually if required for other entities, often with `ConfigToAttributes.apply(this)`.
- Setting ports is a special challenge, and one which the `AttributeSensorAndConfigKey` is particularly helpful for, cf `PortAttributeSensorAndConfigKey` (a subclass), causing ports automatically get assigned from a range and compared with the target `PortSupplied` `location`.

Syntax is as described in the `PortRange` interface. For example, "8080-8099,8800+" will try port 8080, try sequentially through 8099, then try from 8800 until all ports are exhausted.

This is particularly useful on a contended machine (localhost!). Like ordinary configuration, the config is done by the user, and the actual port used is reported back as a `sensor` on the `entity`.

- Validation of config values can be applied by supplying a `Predicate` to the `constraint` of a `ConfigKey` builder. Constraints are tested after an `entity` is initialised and before an `entity` managed. Useful predicates include:
 - `StringPredicates.isNonBlank` : require that a String key is neither null nor empty.
 - `ResourcePredicates.urlExists` : require that a URL that is loadable by Brooklyn. Use this to confirm that necessary resources are available to the `entity`.
 - `Predicates.in` : require one of a fixed set of values.

- `Predicates.containsPattern` : require that a value match a regular expression pattern.

An important caveat is that only constraints on config keys that are on an `entity`'s type hierarchy can be tested automatically. Brooklyn has no knowledge of the true type of other keys until they are retrieved with a

```
config().get(key) .
```

Implementing Sensors

- e.g. HTTP, JMX

Sensors at base entities are often retrieved by feeds which poll the `entity`'s corresponding instance in the real world. The `SoftwareProcess` provides a good example; by subclassing it and overriding the `connectSensors()` method you could wire some example sensors using the following:

```
public void connectSensors() {
    super.connectSensors()

    httpFeed = HttpFeed.builder()
        .entity(this)
        .period(200)
        .baseUrl(mgmtUrl)
        .poll(new HttpPollConfig<Boolean>(SERVICE_UP)
            .onSuccess(HttpValueFunctions.responseCodeEquals(200))
            .onError(Functions.constant(false)))
        .poll(new HttpPollConfig<Integer>(REQUEST_COUNT)
            .onSuccess(HttpValueFunctions.jsonContents("requestCount", Integer.class)))
        .build();
}

@Override
protected void disconnectSensors() {
    super.disconnectSensors();
    if (httpFeed != null) httpFeed.stop();
}
```

In this example (a simplified version of `JBoss7Server`), the url returns metrics in JSON. We report the `entity` as up if we get back an http response code of 200, or down if any other response code or exception. We retrieve the request count from the response body, and convert it to an integer.

Note the first line (`super.connectSensors()`); as one descends into specific convenience subclasses (such as for Java web-apps), the work done by the parent class's overridden methods may be relevant, and will want to be invoked or even added to a resulting list.

For some sensors, and often at compound entities, the values are obtained by monitoring values of other sensors on the same (in the case of a rolling average) or different (in the case of the average of children nodes) entities. This is achieved by policies, described below.

Implementing Effectors

The `Entity` interface defines the sensors and effectors available. The `entity` class provides wiring for the sensors, and the `effector` implementations. In simple cases it may be straightforward to capture the behaviour of the effectors in a simple methods. For example deploying a WAR to a cluster can be done as follows:

This section is not complete. Feel free to [fork](#) the docs and lend a hand.

For some entities, specifically base entities, the implementation of effectors might need other tools (such as SSH), and may vary by `location`, so having a single implementation is not appropriate.

The problem of multiple inheritance (e.g. SSH functionality and `entity` inheritance) and multiple implementations (e.g. SSH versus Windows) is handled in brooklyn using delegates called *drivers*.

In the implementations of `JavaWebApp` entities, the behaviour which the `entity` always does is captured in the `entity` class (for example, breaking deployment of multiple WARs into atomic actions), whereas implementations which is specific to a particular `entity` and driver (e.g. using scp to copy the WARs to the right place and install them, which of course is different among appservers, or using an HTTP or JMX management API, again where details vary between appservers) is captured in a driver class.

Routines which are convenient for specific drivers can then be inherited in the driver class hierarchy. For example, when passing JMX environment variables to Java over SSH, `JavaSoftwareProcessSshDriver` extends

`AbstractSoftwareProcessSshDriver` and parents `JBoss7SshDriver`.

Testing

- Unit tests can make use of `SimulatedLocation` and `TestEntity`, and can extend `BrooklynAppUnitTestSupport`.
- Integration tests and use a `LocalhostMachineProvisioningLocation`, and can also extend `BrooklynAppUnitTestSupport`.

SoftwareProcess Lifecycle

`SoftwareProcess` is the common super-type of most integration components (when implementing in Java).

See `JBoss7Server` and `MySQLNode` for exemplars.

The methods called in a `SoftwareProcess` `entity`'s lifecycle are described below. The most important steps are shown in bold (when writing a new `entity`, these are the methods most often implemented).

- Initial creation (via `EntitySpec` or `YAML`):
 - **no-arg constructor**
 - **init**
 - add locations
 - apply initializers
 - add enrichers
 - add policies
 - add children
 - manages `entity` (so is discoverable by other entities)
- Start:
 - provisions new machine, if the `location` is a `MachineProvisioningLocation`
 - creates new driver
 - **calls `getDriverInterface`**
 - Infers the concrete driver class from the machine-type, e.g. by default it adds "Ssh" before the word "Driver" in "JBoss7Driver".
 - instantiates the driver, **calling the constructor** to pass in the `entity` itself and the machine `location`
 - sets attributes from config (e.g. for ports being used)
 - calls `entity.preStart()`
 - calls `driver.start()`, which:
 - runs pre-install command (see config key `pre.install.command`)
 - uploads install resources (see config keys `files.install` and `templates.install`)
 - **calls `driver.install()`**

- runs post-install command (see config key `post.install.command`)
- **calls** `driver.customize()`
- uploads runtime resources (see config keys `files.runtime` and `templates.runtime`)
- runs pre-launch command (see config key `pre.launch.command`)
- **calls** `driver.launch()`
- runs post-launch command (see config key `post.launch.command`)
- calls `driver.postLaunch()`
- calls `entity.postDriverStart()`, which:
 - calls `entity.waitForEntityStart()` - **waits for** `driver.isRunning()` **to report true**
- **calls** `entity.connectSensors()`
- calls `entity.waitForServiceUp()`
- calls `entity.postStart()`
- Restart:
 - If restarting machine...
 - calls `entity.stop()`, with `stopMachine` set to true.
 - calls `start`
 - restarts children (if configured to do so)
 - Else (i.e. not restarting machine)...
 - calls `entity.preRestart()`
 - calls `driver.restart()`
 - **calls** `driver.stop()`
 - **calls** `driver.launch()`
 - calls `driver.postLaunch()`
 - restarts children (if configured to do so)
 - calls `entity.postDriverStart()`, which:
 - calls `entity.waitForEntityStart()` - **polls** `driver.isRunning()`, waiting for true
 - calls `entity.waitForServiceUp()`
 - calls `entity.postStart()`
- Stop:
 - calls `entity.preStopConfirmCustom()` - aborts if exception.
 - calls `entity.preStop()`
 - stops the process:
 - stops children (if configured to do so)
 - **calls** `driver.stop()`
 - stops the machine (if configured to do so)
 - calls `entity.postStop()`
- Rebind (i.e. when Brooklyn is restarted):
 - **no-arg constructor**
 - reconstitutes `entity` (e.g. setting config and attributes)
 - If `entity` was running...
 - calls `entity.rebind()`; if previously started then:
 - creates the driver (same steps as for start)
 - calls `driver.rebind()`
 - **calls** `entity.connectSensors()`
 - attaches policies, enrichers and persisted feeds
 - manages the `entity` (so is discoverable by other entities)

Service State

Any **entity** can use the standard "service-up" and "service-state" sensors to inform other entities and the GUI about its status.

In normal operation, entities should publish at least one "service not-up indicator", using the `ServiceNotUpLogic.updateNotUpIndicator` method. Each such indicator should have a unique name or input **sensor**. `Attributes.SERVICE_UP` will then be updated automatically when there are no not-up indicators.

When there are transient problems that can be detected, to trigger `ON_FIRE` status **entity** code can similarly set `ServiceProblemsLogic.updateProblemsIndicator` with a unique namespace, and subsequently clear it when the problem goes away. These problems are reflected at runtime in the `SERVICE_PROBLEMS` **sensor**, allowing multiple problems to be tracked independently.

When an **entity** is changing the expected state, e.g. starting or stopping, the expected state can be set using `ServiceStateLogic.setExpectedState`; this expected lifecycle state is considered together with `SERVICE_UP` and `SERVICE_PROBLEMS` to compute the actual state. By default the logic in `computeServiceState` is applied.

For common entities, good out-of-the-box logic is applied, as follows:

- For `SoftwareProcess` entities, lifecycle service state is updated by the framework and a service not-up indicator is linked to the driver `isRunning()` check.
- For common parents, including `AbstractApplication` and `AbstractGroup` subclasses (including clusters, fabrics, etc), the default enrichers analyse children and members to set a not-up indicator (requiring at least one child or member who is up) and a problem indicator (if any children or members are on-fire). In some cases other quorum checks are preferable; this can be set e.g. by overriding the `UP_QUORUM_CHECK` or the `RUNNING_QUORUM_CHECK`, as follows:

```
public static final ConfigKey<QuorumCheck> UP_QUORUM_CHECK = ConfigKeys.newConfigKeyWithDefault(AbstractGroup.UP_QUORUM_CHECK,
    "Require all children and members to be up for this node to be up",
    QuorumChecks.all());
```

Alternatively the `initEnrichers()` method can be overridden to specify a custom-configured **enricher** or set custom config key values (as done e.g. in `DynamicClusterImpl` so that zero children is permitted provided when the initial size is configured to be 0).

For sample code to set and more information on these methods' behaviours, see javadoc in `ServiceStateLogic`, overrides of `AbstractEntity.initEnrichers()` and tests in `ServiceStateLogicTests`.

Notes on Advanced Use

The **enricher** to derive `SERVICE_UP` and `SERVICE_STATE_ACTUAL` from the maps and expected state values discussed above is added by the `AbstractEntity.initEnrichers()` method. This method can be overridden -- or excluded altogether by by overriding `init()` -- and can add enrichers created using the `ServiceStateLogic.newEnricherFromChildren()` method suitably customized using methods on the returned spec object, for instance to look only at members or specify a quorum function (from `QuorumChecks`). If different logic is required for computing `SERVICE_UP` and `SERVICE_STATE_ACTUAL`, use `ServiceStateLogic.newEnricherFromChildrenState()` and `ServiceStateLogic.newEnricherFromChildrenUp()`, noting that the first of these will replace the **enricher** added by the default `initEnrichers()`, whereas the second one runs with a different namespace (unique tag). For more information consult the javadoc on those classes.

Entities can set `SERVICE_UP` and `SERVICE_STATE_ACTUAL` directly. Provided these entities never use the `SERVICE_NOT_UP_INDICATORS` and `SERVICE_PROBLEMS` map, the default enrichers will not override these values.

Entitlements

Brooklyn supports a plug-in system for defining "entitlements" -- essentially permissions.

Any entitlement scheme can be implemented by supplying a class which implements one method on one class:

```
public interface EntitlementManager {
    public <T> boolean isEntitled(@Nullable EntitlementContext context, @Nonnull EntitlementClass<T> entitlementClass, @Nullable T entitlementClassArgument);
}
```

This answers the question who is allowed do what to whom, looking at the following fields:

- `context` : the user who is logged in and is attempting an action (extensions can contain additional metadata)
- `entitlementClass` : the type of action being queried, e.g. `DEPLOY_APPLICATION` or `SEE_SENSOR` (declared in the class `Entitlements`)
- `entitlementClassArgument` : details of the action being queried, such as the `blueprint` in the case of `DEPLOY_APPLICATION` or the `entity` and `sensor` name in the case of `SEE_SENSOR`

To set a custom entitlements manager to apply across the board, simply use:

```
brooklyn.entitlements.global=org.apache.brooklyn.core.mgmt.entitlement.AcmeEntitlementManager
```

The example above refers to a sample manager which is included in the test JARs of Brooklyn, which you can see [here](#), and include in your project by adding the core tests JAR to your `dropins` folder.

There are some entitlements schemes which exist out of the box, so for a simpler setup, see [Operations: Entitlements](#).

There are also more complex schemes which some users have developed, including LDAP extensions which re-use the LDAP authorization support in Brooklyn, allowing permissions objects to be declared in LDAP leveraging regular expressions. For more information on this, ask on IRC or the mailing list, and see

[EntitlementManager](#).

Windows Blueprints

Brooklyn can deploy to Windows servers using WinRM to run commands. These deployments can be expressed in pure [YAML](#), and utilise Powershell to install and manage the software process. This approach is similar to the use of SSH for UNIX-like servers.

About WinRM

WinRM - or *Windows Remote Management* to give its full title - is a system administration service provided in all recent Windows Server operating systems. It allows remote access to system information (provided via WMI) and the ability to execute commands. For more information refer to [Microsoft's MSDN article on Windows Remote Management.aspx](#).

WinRM is available by default in Windows Server, but is not enabled by default. Brooklyn will, in most cases, be able to switch on WinRM support, but this is dependent on your cloud provider supporting a user metadata service with script execution (see [below](#)).

Locations for Windows

You must define a new [location](#) in Brooklyn for Windows deployments. Windows deployments require a different VM image ID to Linux, as well as some other special configuration, so you must have separate Brooklyn locations for Windows and Linux deployments.

In particular, you will most likely want to set these properties on your [location](#):

- `imageId` or `imageNameRegex` - select your preferred Windows Server image from your cloud provider.
- `hardwareId` or `minRam` / `minCores` - since Windows machines generally require more powerful servers, ensure you get a machine with the required specification.
- `useJcloudsSshInit` - this must be set to `false`. Without this setting, jclouds will attempt to connect to the new VMs using SSH, which will fail on Windows Server.
- `templateOptions` - you may also wish to request a larger disk size. This setting is cloud specific; on AWS, you can request a 100GB disk by setting this property to `{mapNewVolumeToDeviceName: ["/dev/sda1", 100, true]}`.

In your [YAML blueprint](#):

```
...
location:
  jclouds:aws-ec2:
    region: us-west-2
    identity: AKA_YOUR_ACCESS_KEY_ID
    credential: <access-key-hex-digits>
    imageNameRegex: Windows_Server-2012-R2_RTM-English-64Bit-Base-.*
    imageOwner: 801119661308
    hardwareId: m3.medium
    useJcloudsSshInit: false
    templateOptions: {mapNewVolumeToDeviceName: ["/dev/sda1", 100, true]}
...
```

Alternatively, you can define a new named [location](#) in `brooklyn.properties` :

```
brooklyn.location.named.AWS\ Oregon\ Win = jclouds:aws-ec2:us-west-2
brooklyn.location.named.AWS\ Oregon\ Win.displayName = AWS Oregon (Windows)
brooklyn.location.named.AWS\ Oregon\ Win.imageNameRegex = Windows_Server-2012-R2_RTM-English-64Bit-Base-.*
```

```
brooklyn.location.named.AWS\ Oregon\ Win.imageOwner = 801119661308
brooklyn.location.named.AWS\ Oregon\ Win.hardwareId = m3.medium
brooklyn.location.named.AWS\ Oregon\ Win.useJcloudsSshInit = false
brooklyn.location.named.AWS\ Oregon\ Win.templateOptions = {mapNewVolumeToDeviceName: ["/dev/sda1", 100, true]}
```

A Sample Blueprint

Creating a Windows VM is done using the `org.apache.brooklyn.entity.software.base.VanillaWindowsProcess` [entity](#) type. This is very similar to `vanillaSoftwareProcess`, but adapted to work for Windows and WinRM instead of Linux. We suggest you read the [documentation for VanillaSoftwareProcess](#) to find out what you can do with this [entity](#).

[Entity](#) authors are strongly encouraged to write Windows Powershell or Batch scripts as separate files, to configure these to be uploaded, and then to configure the appropriate command as a single line that executes given script.

For example - here is a simplified [blueprint](#) (but see [Tips and Tricks](#) below!):

```
name: Server with 7-Zip

location:
  jclouds:aws-ec2:
    region: us-west-2
    identity: AKA_YOUR_ACCESS_KEY_ID
    credential: <access-key-hex-digits>
    imageNameRegex: Windows_Server-2012-R2_RTM-English-64Bit-Base-.*
    imageOwner: 801119661308
    hardwareId: m3.medium
    useJcloudsSshInit: false
    templateOptions: {mapNewVolumeToDeviceName: ["/dev/sda1", 100, true]}

services:
- type: org.apache.brooklyn.entity.software.base.VanillaWindowsProcess
  brooklyn.config:
    templates.preinstall:
      file:///Users/richard/install7zip.ps1: "C:\\install7zip.ps1"
    install.command: powershell -command "C:\\install7zip.ps1"
    customize.command: echo true
    launch.command: echo true
    stop.command: echo true
    checkRunning.command: echo true
    installer.download.url: http://www.7-zip.org/a/7z938-x64.msi
```

The installation script - referred to as `/Users/richard/install7zip.ps1` in the example above - is:

```
$Path = "C:\InstallTemp"
New-Item -ItemType Directory -Force -Path $Path

$Url = "${config['installer.download.url']}"
$Dl = [System.IO.Path]::Combine($Path, "installer.msi")
$WebClient = New-Object System.Net.WebClient
$WebClient.DownloadFile( $Url, $Dl )

Start-Process "msiexec" -ArgumentList '/qn','/i',$Dl -RedirectStandardOutput ( [System.IO.Path]::Combine($Path, "stdout.txt") ) -RedirectStandardError ( [System.IO.Path]::Combine($Path, "stderr.txt") ) -Wait
```

Where security-related operation are to be executed, it may require the use of `credssp` to obtain the correct Administrator privileges: you may otherwise get an access denied error. See the sub-section [How and Why to re-authenticate within a powershell script](#) for more details.

This is only a very simple example. A more complex example can be found in the [Microsoft SQL Server blueprint in the Brooklyn source code](#).

Tips and Tricks

The best practices for other entities (e.g. using [VanillaSoftwareProcess](#)) apply for WinRM as well.

Execution Phases

[Blueprint](#) authors are strongly encouraged to provide an implementation for install, launch, stop and checkRunning. These are vital for the generic effectors such as stopping and restarting the process.

Powershell

Powershell commands can be supplied using config options such as `launch.powershell.command`.

This is an alternative to supplying a standard batch command using config such as `launch.command`. For a given phase, only one of the commands (Powershell or Batch) should be supplied.

Getting the Right Exit Codes

WinRM (or at least the chosen WinRM client!) can return a zero exit code even on error in certain circumstances. It is therefore advisable to follow the guidelines below.

For a given command, write the Powershell or Batch script as a separate multi-command file. Upload this (e.g. by including it in the `files.preinstall` configuration). For the configuration of the given command, execute the file.

When you have a command inside the powershell script which want to report its non zero exiting, please consider adding a check for its exit code after it. Example:

```
& "C:\install.exe"  
If ($lastexitcode -ne 0) {  
    exit $lastexitcode  
}
```

For Powershell files, consider including

```
$ErrorActionPreference = "Stop"
```

at the start of the file. `$ErrorActionPreference` Determines how Windows PowerShell responds to a non-terminating error (an error that does not stop the cmdlet processing) at the command line or in a script, cmdlet, or provider, such as the errors generated by the Write-Error cmdlet. <https://technet.microsoft.com/en-us/library/hh847796.aspx>

See [Incorrect Exit Codes](#) under Known Limitations below.

Executing Scripts From Batch Commands

In a batch command, you can execute a batch file or Powershell file. For example:

```
install.command: powershell -NonInteractive -NoProfile -Command "C:\\install7zip.ps1"
```

Or alternatively:

```
install.command: C:\\install7zip.bat
```

Executing Scripts From Powershell

In a Powershell command, you can execute a batch file or Powershell file. There are many ways to do this (see official Powershell docs). For example:

```
install.powershell.command: "& C:\\install7zip.ps1"
```

Or alternatively:

```
install.powershell.command: "& C:\\install7zip.bat"
```

Note the quotes around the command. This is because the "&" has special meaning in a [YAML](#) value.

Parameterised Scripts

Calling parameterised Batch and Powershell scripts is done in the normal Windows way - see official Microsoft docs. For example:

```
install.command: "c:\\myscript.bat myarg1 myarg2"
```

Or as a Powershell example:

```
install.powershell.command: "& c:\\myscript.ps1 -key1 myarg1 -key2 myarg2"
```

It is also possible to construct the script parameters by referencing attributes of this or other entities using the standard `attributeWhenReady` mechanism. For example:

```
install.command: $brooklyn:formatString("c:\\myscript.bat %s", component("db").attributeWhenReady("datastore.url"))
```

Powershell - Using Start-Process

When you are invoking a command from a powershell script with `start-process` cmdlet, please use the `-wait` and the `-PassThru` arguments. Example `start-process C:\\mycommand -wait -PassThru`

Using `-wait` guarantees that the script process and its children and thus the winrm session won't be terminated until it is finished. `-PassThru` Returns a process object for each process that the cmdlet started. By default, this cmdlet does not generate any output. See <https://technet.microsoft.com/en-us/library/hh849848.aspx>

Rebooting

Where a reboot is required as part of the [entity](#) setup, this can be configured using config like

`pre.install.reboot.required` and `install.reboot.required`. If required, the installation commands can be split between the pre-install, install and post-install phases in order to do a reboot at the appropriate point of the VM setup.

We Strongly recommend to **write blueprints in a way that they do NOT restart automatically windows** and use one of the `pre.install.reboot.required` or `install.reboot.required` parameters to perform restart.

Install Location

Blueprint authors are encouraged to explicitly specify the full path for file uploads, and for paths in their Powershell scripts (e.g. for installation, configuration files, log files, etc).

How and Why to re-authenticate within a powershell script

Some installation scripts require the use of security-related operations. In some environments, these fail by default when executed over WinRM, even though the script may succeed when run locally (e.g. by using RDP to connect to the machine and running the script manually). There may be no clear indication from Windows why it failed (e.g. for MSSQL install, the only clue is a security exception in the installation log).

When a script is run over WinRM, the credentials under which the script are run are marked as 'remote' credentials, which are prohibited from running certain security-related operations. The solution is to obtain a new set of credentials within the script and use those credentials to required commands.

The WinRM client uses Negotiate+NTLM to authenticate against the machine. This mechanism applies certain restrictions to executing commands on the windows host.

For this reason you should enable CredSSP on the windows host which grants all privileges available to the user.

<https://technet.microsoft.com/en-us/library/hh849719.aspx#sectionSection4>

To use `Invoke-Command -Authentication CredSSP` the Windows Machine has to have:

- Up and running WinRM over http. The custom-enable-credssp.ps1 script enables winrm over http because `Invoke-Command` use winrm over http by default. Invoke-Command can be used with `-UseSSL` option but this will lead to modifying powershell scripts. With always enabling winrm over http on the host, blueprint's powershell scripts remain consistent and not depend on the winrm https/http environments. We hope future versions of winrm4j will support CredSSP out of the box and wrapping commands in Invoke-Command will not be needed.
- Added trusted host entries which will use Invoke-Command
- Allowed CredSSP

All the above requirements are enabled in Apache Brooklyn through [brooklyn-server/software/base/src/main/resources/org/apache/brooklyn/software/base/custom-enable-credssp.ps1](#) script which enables executing commands with CredSSP in the general case. The script works for most of the Windows images out there version 2008 and later.

Please ensure that Brooklyn's changes are compatible with your organisation's security [policy](#).

Check Microsoft Documentation for more information about [Negotiate authenticate mechanism on technet.microsoft.com.aspx](#)

Re-authentication also requires that the password credentials are passed in plain text within the script. Please be aware that it is normal for script files - and therefore the plaintext password - to be saved to the VM's disk. The scripts are also accessible via the Brooklyn web-console's activity view. Access to the latter can be controlled via [Entitlements](#).

As an example (taken from MSSQL install), the command below works when run locally, but fails over WinRM:

```
( $driveLetter + "setup.exe") /ConfigurationFile=C:\ConfigurationFile.ini
```

The code below can be used instead (note this example uses Freemarker templating):

```
& winrm set winrm/config/service/auth '@{CredSSP="true"}'
& winrm set winrm/config/client/auth '@{CredSSP="true"}'
#
$pass = '${attribute['windows.password']}'
$secpasswd = ConvertTo-SecureString $pass -AsPlainText -Force
$mycreds = New-Object System.Management.Automation.PSCredential ($($env:COMPUTERNAME + "\"${location.user}"), $s
```



```

ecpasswd)
#
$exitCode = Invoke-Command -ComputerName $env:COMPUTERNAME -Credential $mycreds -ScriptBlock {
    param($driveLetter)
    $process = Start-Process ( $driveLetter + "setup.exe") -ArgumentList "/ConfigurationFile=C:\ConfigurationFile.ini" -RedirectStandardOutput "C:\sqlout.txt" -RedirectStandardError "C:\sqlerr.txt" -Wait -PassThru -NoNewWindow
    $process.ExitCode
} -Authentication CredSSP -ArgumentList $driveLetter
#
exit $exitCode

```

In this example, the `${...}` format is FreeMarker templating. Those sections will be substituted before the script is uploaded for execution. To explain this example in more detail:

- `${attribute['windows.password']}` is substituted for the `entity`'s attribute "windows.password". This (clear-text) password is sent as part of the script. Assuming that HTTPS and NTLM is used, the script will be encrypted while in-flight.
- The `${location.user}` gets (from the `entity`'s machine `location`) the username, substituting this text for the actual username. In many cases, this will be "Administrator". However, on some clouds a different username (with admin privileges) will be used.
- The username and password are used to create a new credential object (having first converted the password to a secure string).
- Credential Security Service Provider (CredSSP) is used for authentication, to pass the explicit credentials when using `Invoke-Command`.

Windows AMIs on AWS

Windows AMIs in AWS change regularly (to include the latest Windows updates). If using the community AMI, it is recommended to use an AMI name regex, rather than an image id, so that the latest AMI is always picked up. If an image id is used, it may fail as Amazon will delete their old Windows AMIs.

If using an image regex, it is recommended to include the image owner in case someone else uploads a similarly named AMI. For example:

```

brooklyn.location.named.AWS\ Oregon\ Win = jclouds:aws-ec2:us-west-2
brooklyn.location.named.AWS\ Oregon\ Win.imageNameRegex = Windows_Server-2012-R2_RTM-English-64Bit-Base-.*
brooklyn.location.named.AWS\ Oregon\ Win.imageOwner = 801119661308
...

```

stdout and stderr in a Powershell script

When calling an executable in a Powershell script, the stdout and stderr will usually be output to the console. This is captured by Brooklyn, and shown in the activities view under the specific tasks.

An alternative is to redirect stdout and stderr to a file on the VM, which can be helpful if one expects sys admins to log into the VM. However, be warned that this would hide the stdout/stderr from Brooklyn's activities view.

For example, instead of running the following:

```
D:\setup.exe /ConfigurationFile=C:\ConfigurationFile.ini
```

The redirect can be achieved by using the `start-process` scriptlet:

```
Start-Process D:\setup.exe -ArgumentList "/ConfigurationFile=C:\ConfigurationFile.ini" -RedirectStandardOutput "C:\sqlout.txt" -RedirectStandardError "C:\sqlerr.txt" -PassThru -Wait
```

The `-ArgumentList` is simply the arguments that are to be passed to the executable, `-RedirectStandardOutput` and `RedirectStandardError` take file locations for the output (if the file already exists, it will be overwritten). The `-PassThru` argument indicates that Powershell should write to the file *in addition* to the console, rather than *instead* of the console. The `-wait` argument will cause the scriptlet to block until the process is complete.

Further details can be found on the [Start-Process documentation page](#) on the Microsoft TechNet site.

Troubleshooting

Much of the [operations troubleshooting guide](#) is applicable for Windows blueprints.

User metadata service requirement

WinRM requires activation and configuration before it will work in a standard Windows Server deployment. To automate this, Brooklyn will place a setup script in the user metadata blob. Services such as Amazon EC2's `Ec2ConfigService` will automatically load and execute this script. If your chosen cloud provider does not support `Ec2ConfigService` or a similar package, or if your cloud provider does not support user metadata, then you must pre-configure a Windows image with the required WinRM setup and make Brooklyn use this image.

If the configuration options `userMetadata` or `userMetadataString` are used on the [location](#), then this will override the default setup script. This allows one to supply a custom setup script. However, if `userMetadata` contains something else then the setup will not be done and the VM may not be accessible remotely over WinRM.

Credentials issue requiring special configuration

When a script is run over WinRM over HTTP, the credentials under which the script are run are marked as 'remote' credentials, which are prohibited from running certain security-related operations. This may prevent certain operations. The installer from Microsoft SQL Server is known to fail in this case, for example. For a workaround, please refer to [How and Why to re-authenticate withing a powershell script](#) above.

WebServiceException: Could not send Message

We detected a `webServiceException` and different `socketException` during deployment of long lasting Application [Blueprint](#) against VcloudDirector.

Launching the [blueprint](#) bellow was giving constantly this type of error on launch step.

```
services:
  type: org.apache.brooklyn.entity.software.base.VanillaWindowsProcess
  brooklyn.config:
    pre.install.command: echo preInstallCommand
    install.command: echo installCommand > C:\install.txt
    post.install.command: echo postInstallCommand
    customize.command: echo customizeCommand
    pre.launch.command: echo preLaunchCommand
    launch.powershell.command: |
      Start-Sleep -s 400
      Write-Host Test Completed
    post.launch.command: echo postLaunchCommand
    checkRunning.command: echo checkRunningCommand
```

```
stop.command: echo stopCommand
```

With series of tests we concluded that on the Vcloud Director environment we were using a restart was happening ~2 minutes after the VM is provisioned. Logging in the host and search for System event of type 1074 in Windows Event Viewer, we found two 1074 events where the second one was

```
The process C:\Windows\system32\winlogon.exe (W2K12-STD) has initiated the restart of computer WIN-XXXX on behalf of user NT AUTHORITY\SYSTEM for the following reason: Operating System: Upgrade (Planned) Reason Code: 0x80020003 Shutdown Type: restart Comment:
```

Normally on other clouds only one restart event is registered and the first time winrm connection is made the Windows VM is ready for use.

For this particular case when you want this second restart to finish we made `waitWindowsToStart` `location` parameter which basically adds additional check assuring the Windows VM provisioning is done.

For example when using `waitWindowsToStart: 5m` `location` parameter, Apache Brooklyn will wait 5 minutes to see if a disconnect occurs. If it does, then it will again wait 5m for the machine to come back up. The default behaviour in Apache Brooklyn is to consider provisioning done on the first successful winrm connection, without waiting for restart.

To determine whether you should use this parameter you should carefully inspect how the image you choose to provision is behaving. If the description above matches your case and you are getting **connection failure message in the middle of the installation process** for your blueprints, a restart probably occurred and you should try this parameter.

Before using this parameter we advice to check whether this is really your case. To verify the behavior check as described above.

AMIs not found

If using the `imageId` of a Windows community AMI, you may find that the AMI is deleted after a few weeks. See [Windows AMIs on AWS](#) above.

VM Provisioning Times Out

In some environments, provisioning of Windows VMs can take a very long time to return a usable VM. If the image is old, it may install many security updates (and reboot several times) before it is usable.

On a VMware vCloud Director environment, the guest customizations can cause the VM to reboot (sometimes several times) before the VM is usable.

This could cause the WinRM connection attempts to timeout. The `location` configuration option `waitForWinRmAvailable` defaults to `30m` (i.e. 30 minutes). This can be increased if required.

Incorrectly prepared Windows template can cause the deployment to time-out expecting an interaction by the user. You can verify if this is the case by RDP to the deployment which is taking too much time to complete. It is recommended to manually deploy a single VM for every newly created Windows template to verify that it can be used for unattended installations and it doesn't wait and/or require an input by the user. See [Windows template settings for an Unattended Installation](#) under Known Limitations below.

Windows log files

Details of the commands executed, and their results, can be found in the Brooklyn log and in the Brooklyn web-console's activity view.

There will also be log files on the Windows Server. System errors in Windows are usually reported in the Windows Event Log -

see <https://technet.microsoft.com/en-us/library/cc766042.aspx> for more information.

Additional logs may be created by some Windows programs. For example, MSSQL creates a log in

`%programfiles%\Microsoft SQL Server\130\Setup Bootstrap\Log\` - for more information see

<https://msdn.microsoft.com/en-us/library/ms143702.aspx>.

Known Limitations

WinRM 2.0 supports encryption mechanisms on top of HTTP. However those are not supported in Apache Brooklyn. For production adoptions please make sure you follow Microsoft Guidelines [https://msdn.microsoft.com/en-us/library/ee309366\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/ee309366(v=vs.85).aspx)

Apache Brooklyn limitations on using WinRM over HTTP and HTTPS

By default Apache Brooklyn is currently using unencrypted HTTP for WinRM communication. It does not support encrypted HTTP for WinRM.

HTTPS is supported but there is no mechanism of specifying which certificates to trust. Currently Apache Brooklyn will accept any certificate used in a HTTPS WinRM connection.

Incorrect Exit Codes

Some limitations with WinRM (or at least the chosen WinRM Client!) are listed below:

Single-line Powershell files

When a Powershell file contains just a single command, the execution of that file over WinRM returns exit code 0 even if the command fails! This is the case for even simple examples like `exit 1` or `thisFileDoesNotExist.exe`.

A workaround is to add an initial command, for example:

```
Write-Host dummy line for workaround
exit 1
```

Direct Configuration of Powershell commands

If a command is directly configured with Powershell that includes `exit`, the return code over WinRM is not respected. For example, the command below will receive an exit code of 0.

```
launch.powershell.command: |
  echo first
  exit 1
```

Direct Configuration of Batch commands

If a command is directly configured with a batch exit, the return code over WinRM is not respected. For example, the command below will receive an exit code of 0.

```
launch.command: exit /B 1
```

Non-zero Exit Code Returned as One

If a batch or Powershell file exits with an exit code greater than one (or negative), this will be reported as 1 over WinRM.

We advise you to use native commands (non-powershell ones) since executing it as a native command will return the exact exit code rather than 1. For instance if you have `installmssql.ps1` script use `install.command: powershell -command "C:\\installmssql.ps1"` rather than using `install.powershell.command: "C:\\installmssql.ps1"`. The first will give you an exact exit code rather than 1

PowerShell "Preparing modules for first use"

The first command executed over WinRM has been observed to include stderr saying "Preparing modules for first use", such as that below:

```
< CLIXML
<Objs Version="1.1.0.1" xmlns="http://schemas.microsoft.com/powershell/2004/04"><Obj S="progress" RefId="0"><TN
RefId="0"><T>System.Management.Automation.PSCustomObject</T><T>System.Object</T></TN><MS><I64 N="SourceId">1</
I64><PR N="Record"><AV>Preparing modules for first use.</AV><AI>0</AI><Nil /><PI>-1</PI><PC>-1</PC><T>Completed
</T><SR>-1</SR><SD> </SD></PR></MS></Obj><Obj S="progress" RefId="1"><TNRef RefId="0" /><MS><I64 N="SourceId">2
</I64><PR N="Record"><AV>Preparing modules for first use.</AV><AI>0</AI><Nil /><PI>-1</PI><PC>-1</PC><T>Comple
ted</T><SR>-1</SR><SD> </SD></PR></MS></Obj></Objs>
```

The command still succeeded. This has only been observed on private clouds (e.g. not on AWS). It could be related to the specific Windows images in use. It is recommended that VM images are prepared carefully, e.g. so that security patches are up-to-date and the VM is suitably initialised.

WinRM executeScript failed: httpLib.BadStatusLine: ''

As described in <https://issues.apache.org/jira/browse/BROOKLYN-173>, a failure has been observed where the 10 attempts to execute the command over WinRM failed with:

```
httpLib.BadStatusLine: ''
```

Subsequently retrying the command worked. It is unclear what caused the failure, but could have been that the Windows VM was not yet in the right state.

One possible workaround is to ensure the Windows VM is in a good state for immediate use (e.g. security updates are up-to-date). Another option is to increase the number of retries, which defaults to 10. This is a configuration option on the machine [location](#), so can be set on the [location](#)'s [brooklyn.properties](#) or in the [YAML](#):

```
execTries: 20
```

Direct Configuration of Multi-line Batch Commands Not Executed

If a command is directly configured with multi-line batch commands, then only the first line will be executed. For example the command below will only output "first":

```
launch.command: |
  echo first
  echo second
```

The workaround is to write a file with the batch commands, have that file uploaded, and execute it.

Note this is not done automatically because that could affect the capture and returning of the exit code for the commands executed.

Install location

Work is required to better configure a default install **location** on the VM (e.g. so that environment variables are set). The installation pattern for linux-based blueprints, of using brooklyn-managed-processes/installs, is not used or recommended on Windows. Files will be uploaded to C:\ if no explicit directory is supplied, which is untidy, unnecessarily exposes the scripts to the user, and could cause conflicts if multiple entities are installed.

Blueprint authors are strongly encourages to explicitly specific directories for file uploads and in their Powershell scripts.

Windows template settings for an Unattended Installation

Windows template needs certain configuration to be applied to prevent windows setup UI from being displayed. The default behavior is to display it if there are incorrect or empty settings. Showing Setup UI will prevent the proper deployment, because it will expect interaction by the user such as agreeing on the license agreement or some of the setup dialogs.

Detailed instruction how to prepare an Unattended installation are provided at <https://technet.microsoft.com/en-us/library/cc722411%28v=ws.10%29.aspx>.

Winrm4j Client

Winrm4j parameters

Check [org.apache.brooklyn.location.winrm.WinRmMachineLocation](#) parameters available for WinRM.

- `host` : Host to connect to (required). Default value `null`
- `port` : WinRM port to use when connecting to the remote machine.
If no port is specified then it defaults to a port depending on the `winrm.useHttps` flag.
- `winrm.useHttps` : The parameter tells the machine sensors whether the winrm port is over https. If the parameter is true then 5986 will be used as a winrm port.
Default value: `false`
- `retriesOfNetworkFailures` : The parameter sets the number of retries for connection failures. If you use high value, consider taking care for the machine's network.
Default value: `4`
- `winrm.useNtlm` : The parameter configures tells the machine sensors whether the winrm port is over https. If the parameter is true then 5986 will be used as a winrm port.
Default value: `true`
- `winrm.computerName` : Windows Computer Name to use for authentication.
Default value: `null`
- `user` : User to connect as
Default value: `null`
- `password` : Password to use to connect.
Default value: `null`
- `waitWindowsToStart` : By default Brooklyn will return the machine immediately after Brooklyn is able to WinRM. Sometimes restart could happen after a Windows VM is provisioned. This could be because of System Upgrade or other. By setting this config key to 60s, 5m or other X Duration of time Brooklyn will wait X amount of time for disconnect to occur. If connection failure occurs it will wait X amount of time for the machine to come up.
Default value: `null`

If there are [location](#) config keys prefixed with `brooklyn.winrm.config.` prefix will be removed and it will be used to instantiate a `org.apache.brooklyn.util.core.internal.winrm.WiRmTool` implementation.

WinRM Connectivity Diagnostics

If you are experiencing problems with a windows [blueprint](#) against a jclouds [location](#) where Apache Brooklyn complains about failing to connect to the IP you should check those things.

1. Apache Brooklyn is using correct username and password
2. Apache Brooklyn can reach the IP of the provisioned machine. WinRM port 5985 or 5986 is also reachable from Apache Brooklyn.
3. Check whether `winRmMachineLocation#getDefaultUserMetadataString(ConfigurationSupportInternal)` is applied on the VM. This script should be passed to the cloud and executed in order to configure WinRM according to Apache Brooklyn requirements for authentication. So far windows startup script are known to be supported on AWS EC2 and VCloud Director. If your cloud doesn't use this script then tune WinRM parameters accordingly.
4. Check whether you use winrm over http or over https.
 - i. If you are using WinRM over http then make sure WinRM service on target VM has `AllowUnencrypted = true`

If the quick list above doesn't help then follow the steps below.

To speed up diagnosing the problem we advice to trigger a deployment with the JcloudsLocation flag `destroyOnFailure: false` so you can check status of the provisioned machine or try later different WinRM parameters with a Apache Brooklyn [BYON Location](#).

After you determined what is the username and the password you can proceed with next steps. *(Notice that for cloud providers which use Auto Generated password will not be logged. For these cases use Java Debug to retrieve or provision a VM manually with the same parameters when using Apache Brooklyn to provision a jclouds [location](#).)*

The first step is to find what is the winrm service configuration on the target host.

1. If you have RDP access or KVM like access to the VM then check the winrm service status with the command bellow. `winrm get winrm/config/service` If you are using http you should have AllowUnencrypted to false. Encryption is supported only over https. Sample output:

```
MaxConcurrentOperations = 4294967295
MaxConcurrentOperationsPerUser = 1500
EnumerationTimeoutms = 240000
MaxConnections = 300
MaxPacketRetrievalTimeSeconds = 120
AllowUnencrypted = true
Auth
  Basic = false
  Kerberos = true
  Negotiate = true
  Certificate = false
  CredSSP = true
  CbtHardeningLevel = Relaxed
DefaultPorts
  HTTP = 5985
  HTTPS = 5986
IPv4Filter = *
IPv6Filter = *
EnableCompatibilityHttpListener = false
EnableCompatibilityHttpsListener = false
CertificateThumbprint
AllowRemoteAccess = true
```

Use an Apache Brooklyn BYON [blueprint](#) to try easily other connection options.

```
location:
  byon:
    hosts:
      - winrm: 10.0.0.1
        user: Administrator
        password: pa55w0rd
        osFamily: windows
    services:
      - type: org.apache.brooklyn.entity.software.base.VanillaWindowsProcess
        brooklyn.config:
          checkRunning.command: echo checkRunning
          install.command: echo installCommand
```

1. Check IP is reachable from Apache Brooklyn instance Check whether `telnet 10.0.0.1 5985` makes successfully a socket.
2. If AllowUnencrypted is false and you are using winrm over http then apply `winrm set winrm/config/service @{AllowUnencrypted="true"}` If jclouds or the cloud provider doesn't support passing `sysprep-specialize-script-cmd` then consider modifying Windows VM Image.
3. Check your username and password. Notice in Windows passwords are case sensitive. Here is how it looks log from a wrong password:


```
INFO: Authorization loop detected on Conduit "{http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd}WinRmPort.ht
tp-conduit" on URL "http://192.168.122.1:5985/wsman" with realm "null"
Oct 21, 2016 10:43:11 AM org.apache.cxf.phase.PhaseInterceptorChain doDefaultLogging
WARNING: Interceptor for {http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd}WinRmService#{http://schemas.dmtf
.org/wbem/wsman/1/wsman.xsd}Create has thrown exception, unwinding now
org.apache.cxf.interceptor.Fault: Could not send Message.
at org.apache.cxf.interceptor.MessageSenderInterceptor$MessageSenderEndingInterceptor.handleMessage(Messag
eSenderInterceptor.java:64)
```

4. When having wrong password you may want to try logging on a different domain This is possible from
`brooklyn.winrm.config.winrm.computerName` [location](#) config.
5. If you want to configure Windows target host with https then check the article [Configuring WINRM for HTTPS](#)
6. If you are still seeing authorization errors then try connecting via winrm with the embedded winrs client. First make sure you have the server in trusted hosts.

Then execute a simple command like

```
winrs -r:10.0.2.15 -unencrypted -u:Administrator -p:pa55w0rd ipconfig
```

Testing YAML Blueprints

Brooklyn provides a selection of test entities which can be used to validate Blueprints via [YAML](#). The basic building block is a `TargetableTestComponent`, which is used to resolve a target. There are two different groups of entities that inherit from `TargetableTestComponent`. The first is structural, which effects how the tests are run, for example by affecting the order they are run in. The second group is validation, which is used to confirm the application is deployed as intended, for example by checking some [sensor](#) value.

Structural test entities include:

- `TestCase` - starts child entities sequentially.
- `ParallelTestCase` - starts child entities in parallel.
- `LoopOverGroupMembersTestCase` - creates a `TargetableTestComponent` for each member of a group.
- `InfrastructureDeploymentTestCase` - will create the specified Infrastructure and then deploy the target [entity](#) specifications there.

Validation test entities include:

- `TestSensor` - perform assertion on a specified [sensor](#).
- `TestEffector` - perform assertion on response to [effector](#) call.
- `TestHttpCall` - perform assertion on response to specified HTTP GET Request.
- `TestSshCommand` - test assertions on the result of an ssh command on the same machine as the target [entity](#).
- `TestWinrmCommand` - test assertions on the result of a WinRM command on the same machine as the target [entity](#).
- `TestEndpointReachable` - assert that a TCP endpoint is reachable. The endpoint can be in a number of different formats: a string in the form of `ip:port` or URI format; or a `com.google.common.net.HostAndPort` instance; or a `java.net.URI` instance; or a `java.net.URL` instance.

`TargetableTestComponents` can be chained together, with the target being inherited by the components children. For example, a `ParallelTestCase` could be created that has a `TestHttpCall` as a child. As long as the `TestHttpCall` itself does not have a target, it will use the target of it's parent, `ParallelTestCase`. Using this technique, we can build up complex test scenarios.

The following sections provide details on each test [entity](#) along with examples of their use.

- [Test Entities](#)
- [Example Tests](#)

Test Entities

Structural Test Entities

TestCase

The `TestCase` entity acts as a container for a list of child entities which are started *sequentially*.

```
- type: org.apache.brooklyn.test.framework.TestCase
brooklyn.children:
- type: org.apache.brooklyn.entity.database.mysql.MySqlNode
  ...
- type: org.apache.brooklyn.test.framework.TestSensor
  ...
```

This can be used to enforce a strict ordering, for example ensuring a `sensor` has a certain value before attempting to invoke an `effector`.

Timeouts on child entities should be set relative to the completion of the preceding entity.

The `ParallelTestCase` entity can be added as a child to run a subset of entities in parallel as a single step.

ParallelTestCase

The `ParallelTestCase` entity acts as a container for a list of child entities which are started in *parallel*.

```
- type: org.apache.brooklyn.test.framework.ParallelTestCase
brooklyn.children:
- type: org.apache.brooklyn.entity.database.mysql.MySqlNode
  ...
- type: org.apache.brooklyn.entity.webapp.ControlledDynamicWebAppCluster
  ...
```

This can be used to run a subset of entities in parallel as a single step when nested under a `TestCase` entity.

Timeouts on child entities should be set relative to the start of the `ParallelTestCase`.

LoopOverGroupMembersTestCase

The `LoopOverGroupMembersTestCase` entity is configured with a target group and a test specification. For each member of the targeted group, the test case will create a `TargetableTestComponent` entity from the supplied test specification and set the components target to be the group member.

```
- type: org.apache.brooklyn.test.framework.LoopOverGroupMembersTestCase
brooklyn.config:
  target: $brooklyn:entity("infrastructure").component("child", "DockerHosts")
test.spec:
  $brooklyn:entitySpec:
    type: org.apache.brooklyn.test.framework.TestSensor
    ...
```

Parameters

- `target` - group who's members are to be tested, specified via DSL. For example, `$brooklyn:entity("tomcat")`. See also the `targetId` parameter.
- `targetId` - alternative to the `target` parameter which wraps the DSL component lookup requiring only the `id` be supplied. For example, `tomcat`. Please note, this must point to a group.
- `test.spec` - The TargetableTestComponent to create for each child.

InfrastructureDeploymentTestCase

The `InfrastructureDeploymentTestCase` will first create and deploy an infrastructure from the `infrastructure.deployment.spec` config. It will then retrieve a deployment `location` by getting the value of the `infrastructure.deployment.location.sensor` `sensor`. It will then create and deploy all entities from the `infrastructure.deployment.spec` config to the deployment `location`.

```
- type: org.apache.brooklyn.test.framework.InfrastructureDeploymentTestCase
brooklyn.config:
  infrastructure.deployment.location.sensor: entity.dynamicLocation
  infrastructure.deployment.spec:
    $brooklyn:entitySpec:
      - type: docker-cloud-calico
      ...
  infrastructure.deployment.entity.specs:
    - $brooklyn:entitySpec:
        type: org.apache.brooklyn.entity.software.base.VanillaSoftwareProcess
      ...
```

Parameters

- `infrastructure.deployment.spec` - the infrastructure to be deployed.
- `infrastructure.deployment.entity.specs` - the entities to be deployed to the infrastructure
- `infrastructure.deployment.location.sensor` - the name of the `sensor` on the infrastructure to retrieve the deployment `location`

Validation Test Entities

TestSensor

The `TestSensor` `entity` performs an assertion on a specified sensors value.

```
- type: org.apache.brooklyn.test.framework.TestSensor
name: Check tomcat isUp
brooklyn.config:
  target: $brooklyn:entity("tomcat")
  sensor: service.isUp
  timeout: 10m
  assert:
    - equals: true
```

Parameters

- `target` - `entity` whose `sensor` will be tested, specified via DSL. For example, `$brooklyn:entity("tomcat")`. See also

the `targetId` parameter.

- `targetId` - alternative to the `target` parameter which wraps the DSL component lookup requiring only the `id` be supplied. For example, `tomcat` .
- `sensor` - `sensor` to evaluate. For example `service.isUp` .
- `timeout` - duration to wait on assertion to return a result. For example `10s` , `10m` , etc
- `assert` - assertion to perform on the specified `sensor` value. See section on assertions below.

Tip

If the `TestSensor` is wrapped within a `TestCase` , `ParallelTestCase` Or `LoopOverGroupMembersTestCase` that set the target, **you don't need to specify the target**, unless you want to test a different `entity`.

TestEffector

The `TestEffector` `entity` invokes the specified `effector` on a target `entity`. If the result of the `effector` is a String, it will then perform assertions on the result.

```
- type: org.apache.brooklyn.test.framework.TestEffector
  name: Invoke Deploy Effector
  brooklyn.config:
    target: $brooklyn:entity("tomcat")
    effector: deploy
    timeout: 5m
    params:
      url: http://search.maven.org/remotecontent?filepath=org/apache/brooklyn/example/brooklyn-example-hello-world-sql-webapp/0.8.0-incubating/brooklyn-example-hello-world-sql-webapp-0.8.0-incubating.war
      targetName: newcontext
```

Parameters

- `target` - `entity` whose `effector` will be invoked, specified via DSL. For example, `$brooklyn:entity("tomcat")` . See also the `targetId` parameter.
- `targetId` - alternative to the `target` parameter which wraps the DSL component lookup requiring only the `id` be supplied. For example, `tomcat` .
- `timeout` - duration to wait on the `effector` task to complete. For example `10s` , `10m` , etc
- `effector` - `effector` to invoke, for example `deploy` .
- `params` - parameters to pass to the `effector`, these will depend on the `entity` and `effector` being tested. The example above shows the `url` and `targetName` parameters being passed to Tomcats `deploy` `effector`.
- `assert` - assertion to perform on the returned result. See section on assertions below.

Tip

If the `TestEffector` is wrapped within a `TestCase` , `ParallelTestCase` Or `LoopOverGroupMembersTestCase` that set the target, **you don't need to specify the target**, unless you want to test a different `entity`.

TestHttpCall

The `TestHttpCall` `entity` performs a HTTP GET on the specified URL and performs an assertion on the response.

```
- type: org.apache.brooklyn.test.framework.TestHttpCall
  name: Check HTTP Response Status Code
  brooklyn.config:
```

```
url: $brooklyn:entity("tomcat").attributeWhenReady("webapp.url")
timeout: 60s
applyAssertionTo: status
assert:
- isEqualTo: 200
```

Parameters

- `url` - URL to perform GET request on, this can use DSL for example `$brooklyn:entity("tomcat").attributeWhenReady("webapp.url")`.
- `timeout` - duration to wait on a HTTP response. For example `10s`, `10m`, etc
- `applyAssertionTo` - The field to apply the assertion to. For example `status`, `body`
- `assert` - assertion to perform on the response. See section on assertions below.

Tip

If the `TestHttpRequest` is wrapped within a `TestCase`, `ParallelTestCase` Or `LoopOverGroupMembersTestCase` that set the target, **you don't need to specify the target**, unless you want to test a different `entity`.

TestSshCommand

The `TestSshCommand` runs a command on the host of the target `entity`. The script is expected not to run indefinitely, but to return a result (process exit code), along with its standard out and error streams, which can then be tested using assertions. If no assertions are explicitly configured, the default is to assert a non-zero exit code.

Either a bash command may be provided in the `YAML`, or a URL for a script which will be executed.

```
services:
- type: org.apache.brooklyn.test.framework.TestCase
  name: testcase1
  brooklyn.children:
  - type: org.apache.brooklyn.entity.webapp.tomcat.TomcatServer
    id: testprocess

  - type: org.apache.brooklyn.test.framework.TestSshCommand
    name: Check tomcat process running with ps
    brooklyn.config:
      targetId: testprocess
      command: ps -ef
      assert.status:
        equals: 0
      assert.out:
        contains: tomcat
      assert.err:
        isEmpty: true

  - type: org.apache.brooklyn.test.framework.TestSshCommand
    name: Check hello world script
    brooklyn.config:
      targetId: testprocess
      download.url: https://github.com/apache/brooklyn-docs/raw/master/guide/blueprints/test/example_yaml/entities/script1.sh
      assert.status:
        equals: 0
      assert.out:
        equals: hello world
      assert.err:
        isEmpty: true
```

Parameters

- `command` - The shell command to execute. (This and `downloadUrl` are mutually exclusive.)
- `downloadUrl` - URL for a script to download and execute. (This and `command` are mutually exclusive.)
- `shell.env` - Map of environment variables to be set.
- `scriptDir` - if `downloadUrl` is used. The directory on the target host where downloaded scripts should be copied to.
- `runDir` - the working directory where the command or script will be executed on the target host.
- `assertStatus` - Assertions on the exit code of the command or script. See section on assertions below.
- `assertOut` - Assertions on the standard output of the command as a String.
- `assertErr` - Assertions on the standard error of the command as a String.

Tip

If the `TestSshCommand` is wrapped within a `TestCase`, `ParallelTestCase` Or `LoopOverGroupMembersTestCase` that set the target, **you don't need to specify the target**, unless you want to test a different [entity](#).

Assertions

The following conditions are provided by those test entities above that include assertions

- `isNull` - asserts that the actual value is `null`.
- `notNull` - asserts that the actual value is NOT `null`.
- `isEqualTo` - asserts that the actual value equals an expected value.
- `equalTo` - a synonym for `isEqualTo`
- `equals` - a synonym for `isEqualTo`
- `notEqual` - asserts that the actual value does not equal the expected value.
- `matches` - asserts that the actual value matches a [regex pattern](#), for example `".*hello.*"`. Note that regular expressions follow the Java defaults, e.g. for [line terminators](#). One can use `"(?m)"` for multi-line mode (so that `^` and `$` also match just after/before line terminators, rather than just the start/end of the entire input sequence). The dotall mode, set with `"(?s)"`, can also be useful (so that `.` matches any character, including line terminators).
- `containsMatch` - asserts that the value contains a string that matches a regex. It follow the behaviour of the [Matcher.find\(\) method](#)). This can be useful for simplifying matching in multi-line input.
- `contains` - asserts that the actual value contains the supplied value
- `isEmpty` - asserts that the actual value is an empty string
- `notEmpty` - asserts that the actual value is a non empty string
- `hasTruthValue` - asserts that the actual value has the expected interpretation as a boolean
- `greaterThan` - asserts that the actual value is greater than the expected value according to Java's [Comparable](#) interface. Actual and expected must be instances of the same type and implement `Comparable`.
- `lessThan` - asserts that the actual value is less than the expected value according to Java's [Comparable](#) interface. Actual and expected must be instances of the same type and implement `Comparable`.

Assertions may be provided as a simple map:

```
assert:
  contains: 2 users
  matches: .*[\d]* days.*
```

If there is the need to make multiple assertions with the same key, the assertions can be specified as a list of such maps:

```
assert:
```

- contains: 2 users
- contains: 2 days

Example Tests

Introduction

This section describes some simple tests based on the [Getting Started](#) example [blueprint](#):

```
name: My Web Cluster

location:
  jclouds:aws-ec2:
    identity: ABCDEFGHIJKLMNOPQRST
    credential: s3cr3tsq1rr3ls3cr3tsq1rr3ls3cr3tsq1rr3l

services:
- type: org.apache.brooklyn.entity.webapp.ControlledDynamicWebAppCluster
  name: My Web
  id: webappcluster
  brooklyn.config:
    wars.root: http://search.maven.org/remotecontent?filepath=org/apache/brooklyn/example/brooklyn-example-hello-world-sql-webapp/0.8.0-incubating/brooklyn-example-hello-world-sql-webapp-0.8.0-incubating.war
    java.sysprops:
      brooklyn.example.db.url: >
        $brooklyn:formatString("jdbc:%s?s?user=%s&password=%s",
          component("db").attributeWhenReady("datastore.url"),
          "visitors", "brooklyn", $brooklyn:external("brooklyn-demo-sample", "hidden-brooklyn-password"))
- type: org.apache.brooklyn.entity.database.mysql.MySqlNode
  name: My DB
  id: db
  brooklyn.config:
    creation.script.password: $brooklyn:external("brooklyn-demo-sample", "hidden-brooklyn-password")
    datastore.creation.script.url: https://bit.ly/brooklyn-visitors-creation-script
```

The following sections contain [yaml](#) snippets that be appended to the list of services in the [blueprint](#) above, a complete [blueprint](#) is also provided [below](#).

Note that unless otherwise specified the following tests are executed in parallel with the deployment of the application services, all `timeout` values are set accordingly.

Sensor Test

Demonstrates the following [sensor](#) assertion:

- asserts that the `webappcluster` [entity](#) `service.isUp` [sensor](#) is `true` within 10 minutes of [blueprint](#) being deployed.

```
- type: org.apache.brooklyn.test.framework.TestSensor
  name: Check webappcluster isUp
  brooklyn.config:
    targetId: webappcluster
    sensor: service.isUp
    timeout: 10m
  assert:
    - equals: true
```

HTTP Call Tests

Demonstrates the following HTTP Call assertions against the specified `url`, which in these examples are being built from the `webappcluster` entities `host.address` and `proxy.http.port` sensors (the tester having flexibility in how the test URL is to be constructed):

- asserts the response status code is 200 within 10 minutes of the `blueprint` being deployed.
- asserts the response body matches the regex `(?s).*Br[o]{2}klyn Deployed.*` within 10 minutes of the `blueprint` being deployed. Note the presence of the `(?s)` `dotall` flag to test a multiline response.

```
- type: org.apache.brooklyn.test.framework.TestHttpCall
name: Check HTTP Response Status Code
brooklyn.config:
  url: >
    $brooklyn:formatString("http://%s:%s/newcontext/",
    $brooklyn:component("webappcluster").attributeWhenReady("host.address"),
    $brooklyn:component("webappcluster").attributeWhenReady("proxy.http.port"))
  timeout: 10m
  applyAssertionTo: status
  assert:
    - equals: 200
- type: org.apache.brooklyn.test.framework.TestHttpCall
name: Check HTTP Response Body
brooklyn.config:
  url: >
    $brooklyn:formatString("http://%s:%s/newcontext/",
    $brooklyn:component("webappcluster").attributeWhenReady("host.address"),
    $brooklyn:component("webappcluster").attributeWhenReady("proxy.http.port"))
  timeout: 10m
  applyAssertionTo: body
  assert:
    - matches: "(?s).*Br[o]{2}klyn Deployed.*"
```

Effector Test (via TestCase entity)

This `TestEffector` example demonstrates the use of the `TestCase` and `TestSensor` entities to ensure the service has started before invoking an `effector` using the `TestEffector` entity.

- `TestCase` entity starts its children sequentially
 - asserts that the `webappcluster` entity `service.isUp` sensor is `true` within 10 minutes of the parent `TestCase` entity starting. Blocks start of the next child until it obtains a result (or times out).
 - `deploy` effector invoked to deploy war to a `newcontext` with a 5 minute timeout to allow completion of the deploy task.
 - asserts `/newcontext` url returns a HTTP status code 200 within 5 minutes of the `effector` being invoked (Note that this timeout is relative to the preceding test entity as they are being sequentially run as children of a `TestCase` entity).

```
- type: org.apache.brooklyn.test.framework.TestCase
name: Check Deploy Effector
brooklyn.children:
- type: org.apache.brooklyn.test.framework.TestSensor
name: Check webappcluster isUp
brooklyn.config:
  targetId: webappcluster
  sensor: service.isUp
  timeout: 10m
  assert:
    - equals: true
- type: org.apache.brooklyn.test.framework.TestEffector
name: Invoke Deploy Effector
brooklyn.config:
  targetId: webappcluster
  effector: deploy
```

```

    timeout: 5m
    params:
      url: http://search.maven.org/remotecontent?filepath=org/apache/brooklyn/example/brooklyn-example-hello-
world-sql-webapp/0.8.0-incubating/brooklyn-example-hello-world-sql-webapp-0.8.0-incubating.war
      targetName: newcontext
- type: org.apache.brooklyn.test.framework.TestHttpCall
  name: Check Deployed Webapp Status Code
  brooklyn.config:
    timeout: 5m
    url: >
      $brooklyn:formatString("http://%s:%s/newcontext/",
        $brooklyn:entity("webappcluster").attributeWhenReady("host.address"),
        $brooklyn:entity("webappcluster").attributeWhenReady("proxy.http.port"))
    applyAssertionTo: status
    assert:
      - isEqualTo: 200

```

Full Example

A sample [blueprint](#) containing all the tests described above is available [here](#).

This [blueprint](#) will deploy the [Getting Started](#) application and run all of the test entities, which if successful should appear in the web console as follows.

The screenshot shows the Brooklyn web console interface. At the top, there's a navigation bar with 'Home', 'Applications' (highlighted), 'Catalog', 'Script', and a help icon. The main content area is titled 'Applications' and shows a list of applications on the left and a detailed view of the 'Getting Started Blueprint Test' on the right.

Applications List (Left):

- Getting Started Blueprint Test
 - Check Deploy Effector
 - Check Deployed Webapp Status Code
 - Check webappcluster isUp
 - Invoke Deploy Effector
 - Check HTTP Response Body
 - Check HTTP Response Status Code
 - My DB
 - My Web
 - Cluster of TomcatServer (JcloudsLocat
 - quarantine
 - TomcatServer:Knhs
 - NginxController:yla0
 - TomcatServer:Knhs

Getting Started Blueprint Test Detail (Right):

- Summary** (selected), Sensors, Effectors, Policies, Activity, Advanced
- Status** (dropdown menu)

Status	RUNNING
Service Up	true
Type	org.apache.brooklyn.entity.stock.BasicApplication
ID	kDfBgKmx
- Blueprint** (dropdown menu)
- Config** (dropdown menu)

Ansible in YAML Blueprints

This guide describes how Brooklyn entities can be created using the Ansible infrastructure management tool (ansible.com). At present Brooklyn provides basic support for Ansible, operating in a 'masterless' mode. Comments on this support and suggestions for further development are welcome.

This guide assumes you are familiar with the basics of [creating YAML blueprints](#).

- [About Ansible](#)
- [Creating Blueprints with Ansible](#)

About Ansible

What you need to know about Ansible

Ansible is a deployment tool designed to work in an agent-less manner, normally performing its operations on a node over SSH from some central administrating node. Brooklyn can deploy software via Ansible on one of its managed nodes, by first installing Ansible on the node itself and then using Ansible to deploy the required software.

A 'Playbook' in Ansible is a specification of the configuration and deployment of a system. Playbooks are expressed in YAML format, and contain a number of 'plays', which are in turn lists of tasks to carry out to achieve the desired configuration on the system. 'Roles' are pre-written modular collections of tasks, and can be included in playbooks.

Ansible comes with built-in support for many software systems, and has a community repository of roles exists at <https://galaxy.ansible.com>.

How Brooklyn interacts with Ansible

Brooklyn provides a Ansible **entity** type. An **entity** of this type can be specified in a **blueprint** in order to provision the node through Ansible. The Ansible **entity** will download Ansible and install it on the node. The **entity** type supports the configuration of Ansible playbooks to download, or write inline in the **blueprint**, for simple playbooks. Configuration values for the playbooks can be supplied in the **blueprint**.

Brooklyn will deploy the software specified in the playbook when the **entity** starts. In addition, an **effector** is provided on the **entity** that supports general purpose Ansible instructions.

Creating Blueprints with Ansible

To write a [blueprint](#) to use Ansible with Brooklyn it will help to have a degree of familiarity with Ansible itself. In the sections below, when the Brooklyn configuration is described, the underlying Ansible operation is also noted briefly, for clarity for readers who know Ansible.

To manage a node with Ansible, create a [blueprint](#) containing a service of type `org.apache.brooklyn.entity.cm.ansible.AnsibleEntity` and define and minimum the `playbook` value, and one or other of `playbook.url` or `playbook.yaml`. You must also define the `service.name` that will be tested in order to determine if the [entity](#) is running successfully.

For example:

```
name: myweb
location: ...
services:
- type: org.apache.brooklyn.entity.cm.ansible.AnsibleEntity
  id: apache
  name: apache
  service.name: apache2
  playbook: apache-playbook
  playbook.url: http://myhost/projectX/apache-playbook.yaml
```

This example specifies that Brooklyn should use Ansible to download the playbook from the repository on "myhost". The playbook contains the instructions to install the Apache web server. To start the [entity](#), Brooklyn will use Ansible's "ansible-playbook" command to run the playbook, which will bring up the web server.

Lifecycle of AnsibleEntity

The start [effector](#) applies the playbook and verifies that it has started the software correctly by checking the service defined as `service.name` is running. This can be customized, see `ansible.service.start` configuration below.

The stop [effector](#) will stop the service `service.name`. Again, this can be customized, with `ansible.service.stop`.

The restart [effector](#) will apply stop and then start.

Configuration of AnsibleEntity

The `playbook` configuration key names the top level list of states that will be applied using Ansible.

This configuration key is mandatory.

The playbook must be defined using one or other (both together are not permitted) of `playbook.yaml` or `playbook.url`.

The former allows the playbook content to be defined inline within the [blueprint](#), using the normal [YAML](#) format of an Ansible playbook. The latter obtains the playbook from an external URL.

The `ansible.service.start` configuration key allows the [blueprint](#) author to override the command used by default to verify that the service `service.name` is running (or to start it, if the playbook did not specify it should run by default). The default value is:

```
sudo ansible localhost -c local -m service -a "name=<service.name> state=started"
```

Similarly the `ansible.service.stop` configuration key permits override of the instruction used to get Ansible to stop the service, by default

```
sudo ansible localhost -c local -m service -a "name=<service.name> state=stopped"
```

The `ansible.service.checkPort` configuration key allows the user to override the mechanism used to check that the service `service.name` is operating. By default Brooklyn checks that the service process is running. However, if the service is one that listens on a particular port, this configuration key allows the [blueprint](#) author to instruct Brooklyn to check that the port is being listened on, using the Ansible `wait_for` module. The value of the key is the port number to check.

The `ansible.vars` configuration key allows the [blueprint](#) author to provide [entity](#)-specific values for configuration variables used in the playbook, so that one playbook can be used by multiple entities, each customized appropriately. The value of `ansible.vars` is an arbitrary block of [YAML](#) configuration that will be applied to the playbook using Ansible's `--extra-vars` mechanism, as described in the Ansible [documentation](#). For example, if the playbook in the example above contained configuration such as:

```
- hosts: all
  vars:
    http_port: 80
    max_clients: 200
    remote_user: root
  tasks:
    ...
```

then to change the port that the webserver in the example above runs on, it would be possible to define the following in the [blueprint](#):

```
name: myweb
location: ...
services:
  - type: org.apache.brooklyn.entity.cm.ansible.AnsibleEntity
    id: apache
    name: apache
    service.name: apache2
    playbook: apache-playbook
    playbook.url: http://myhost/projectX/apache-playbook.yaml
    ansible.vars:
      http_port: 8080
```

ansibleCall Effector

The Ansible [entity](#) includes a general purpose Ansible [effector](#), `ansibleCommand`, which permits execution of Ansible commands via `ansible`. It contains a two parameters:

1. `module` specifies the Ansible module to invoke. The default is "command".
2. `args` specifies the argument data for the Ansible module. For example, to download an additional file for the webserver, the command could be invoked with the following arguments. (For convenience this example uses the client CLI, "br", but the [effector](#) could be invoked by any applicable means, e.g. via the web UI or REST API.)

```
$ br app myweb ent apache effector ansibleCommand invoke \
```

```
-P module=shell -P args='curl http://myhost:8080/additional.html > /var/www/html/additional.html'
```

Roles and Multi-Playbook Installations

There is no specific configuration in AnsibleEntity for Ansible Roles, or to install multiple playbooks. However, the installation of roles or multiple playbooks can be carried out first by taking advantage of Brooklyn's SameServerEntity. The installation step can be applied in one child of the same server entity, while the AnsibleEntity can operate under the second child. It will typically be necessary to delay the start of the AnsibleEntity until the first child has carried out whatever preparation is required. The examples below illustrate the concept (with just one playbook, for brevity).

One way to achieve this, as with any Brooklyn entity, is to use the idiom of making it a child of a BasicApplication with a start latch waiting on the first child, such as in the following example, which installs the standalone Tomcat example, with its playbook and roles, from the "Ansible Examples" Github site. (Note, this is designed for installation on Redhat/Centos 6.) The first child in the SameServerEntity downloads and unpacks the Ansible examples. This might also install standalone Ansible roles, or whatever other resources the AnsibleEntity might require. The second child uses `attributeWhenReady` to block until the first is ready, before starting the AnsibleEntity to apply the desired playbook.

```
name: multi
location:
  red1
services:
- serviceType: brooklyn.entity.basic.SameServerEntity
  name: Entities
  brooklyn.children:

  - serviceType: org.apache.brooklyn.entity.cm.ansible.AnsibleEntity
    id: bootstrap
    service.name: crond
    playbook: bootstrap
    playbook.yaml: |
      ---
      - hosts: localhost
        tasks:
        - shell: printf "[tomcat-servers]\nlocalhost ansible_connection=local\n" >> /etc/ansible/hosts
        - file: path=/etc/ansible/playbooks state=directory mode=0755
        - get_url: url=https://github.com/ansible/ansible-examples/archive/master.zip dest=/tmp/master.zip mode=0440
        - command: unzip -o -d /etc/ansible/playbooks /tmp/master.zip

  - serviceType: org.apache.brooklyn.entity.stock.BasicApplication
    latch.start: $brooklyn:entity("bootstrap").attributeWhenReady("service.isUp")
    brooklyn.children:
    - type: org.apache.brooklyn.entity.cm.ansible.AnsibleEntity
      name: test
      service.name: tomcat
      playbook: tomcat
      playbook.yaml: |
        ---
        - hosts: localhost
        - include: /etc/ansible/playbooks/ansible-examples-master/tomcat-standalone/site.yml
        vars:
          http_port: 8080
          https_port: 8443
          admin_username: admin
          admin_password: secret
```

An alternative to the above is to use Ansible itself to do the waiting, as in the variant below, which uses AnsibleEntity itself in the first SameServerEntity child, to install the required material. In the second child, which is simply an AnsibleEntity rather than a BasicApplication, Ansible's `wait_for` operation is used as the first step in the playbook, to block the remaining steps in its playbook until the first is complete.

```
name: multi
location:
  red1
services:
- serviceType: brooklyn.entity.basic.SameServerEntity
```



```

name: Entities
brooklyn.children:

- serviceType: org.apache.brooklyn.entity.cm.ansible.AnsibleEntity
  id: bootstrap
  service.name: crond
  playbook: bootstrap
  playbook.yaml: |
    ---
    - hosts: localhost
      tasks:
        - command: rm -f /tmp/bootstrap.done
        - shell: printf "[tomcat-servers]\nlocalhost ansible_connection=local\n" >> /etc/ansible/hosts
        - file: path=/etc/ansible/playbooks state=directory mode=0755
        - get_url: url=https://github.com/ansible/ansible-examples/archive/master.zip dest=/tmp/master.zip mo
de=0440
        - command: unzip -o -d /etc/ansible/playbooks /tmp/master.zip
        - file: path=/tmp/bootstrap.done state=touch

- serviceType: org.apache.brooklyn.entity.cm.ansible.AnsibleEntity
  name: test
  service.name: tomcat
  playbook: tomcat
  playbook.yaml: |
    ---
    - tasks:
      - wait_for: path=/tmp/bootstrap.done
      include: /etc/ansible/playbooks/ansible-examples-master/tomcat-standalone/site.yml
      vars:
        http_port: 8080
        https_port: 8443
        admin_username: admin
        admin_password: secret

```

Chef in YAML Blueprints

This guide describes how Brooklyn entities can be easily created from Chef cookbooks. As of this writing (May 2014) some of the integration points are under active development, and comments are welcome. A plan for the full integration is online [here](#).

This guide assumes you are familiar with the basics of [creating YAML blueprints](#).

- [About Chef](#)
- [Creating Blueprints from Chef](#)
- [Writing Chef for Blueprints](#)
- [Advanced Chef Integration](#)

About Chef

What you need to know about Chef

Chef works in two different modes, *server* and *solo*. *Server* is where the Chef client talks to a central server to retrieve information about its roles, policies and cookbooks (where a cookbook defines how to install and configure a particular piece of software). With *solo*, the client works in isolation, therefore its configuration and cookbooks must be supplied by another means.

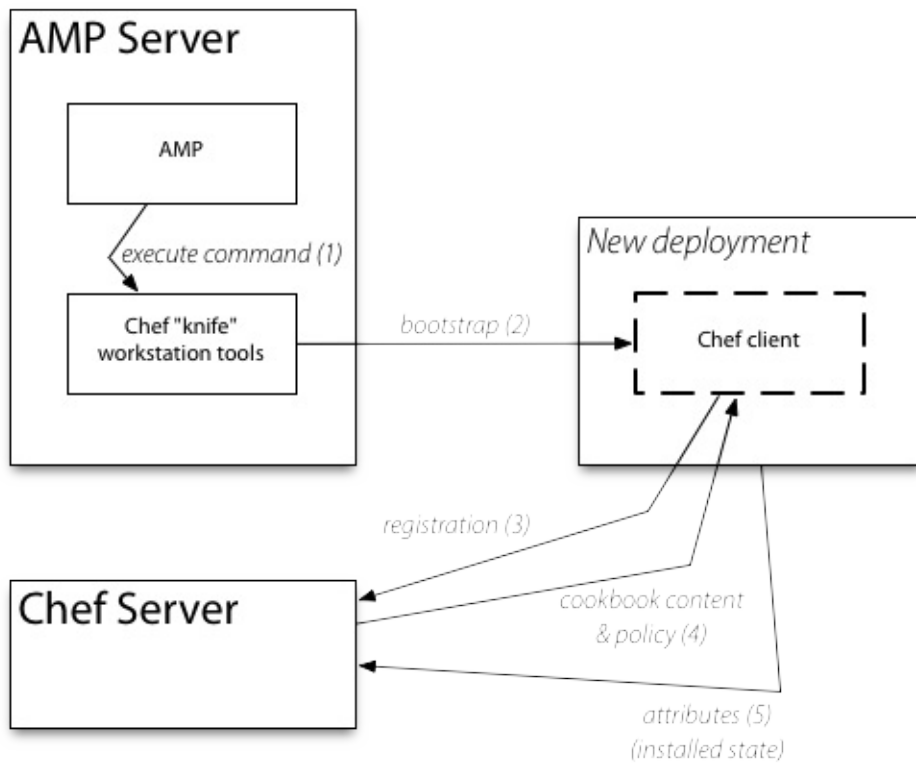
Chef *client* is the Chef agent. This is a Ruby application which is installed on each and every managed host. When invoked in server mode, it will contact the Chef server to check for updates to cookbooks and *policy*; it then "runs" the recipes in its run lists, to converge the machine to a known state. In solo mode, it reads the locally-maintained cookbooks and policies. The client may be run as a daemon that checks the server regularly, or it could merely be run manually when required.

The *policy* is a set of rules on the Chef server. A client starts with a set of *attributes*, which could be as simple as its name and a recipe runlist, or which may involve a more complex set of attributes about how it is to be configured. The client then augments this with auto-detected metadata - a tool called `ohai` is run that collects detailed information about the host. Next, the *policy* on the server modifies these attributes - overriding some, setting defaults for others - to produce a final set of attributes. It is these which are the input to the recipes. Finally, the attributes are uploaded to the server where they are stored as metadata for the node, where they can be inspected and modified by the system operator.

Also of interest is `knife`, which is the workstation toolkit for Chef. Typically this would be installed on the operation engineer's workstation, where it would be used to interact with the Chef server and clients. Of particular interest to us is the *bootstrap* operation, which is used for setting up new Chef clients - given a virtual machine, it will install the Chef client on it, configure it with enough information to find the Chef server and performs its first run, and then kicks off the Chef client for the first time.

There is often a preconception about how a Chef client is bootstrapped; mistakenly, there is the belief that the `knife` tool configures the Chef server with information about the client, and the client finds out about itself from the server. This is not the case - the bootstrap operation does not involve `knife` talking to the server. Instead, `knife` packages up all of the required information and sends it to the client - the client will then introduce itself to the server, passing on its configuration.

This diagram summarises the interaction between Brooklyn, the new node, and the various Chef tools. Note that there is no interaction between the Apache Brooklyn Server and the Chef Server.



How Brooklyn interacts with Chef

Brooklyn understands both the *server* and *solo* modes of operation. Server mode utilises the `knife` toolkit, and therefore `knife` must be installed onto the Apache Brooklyn server and configured appropriately. Solo mode does not have any special requirements; when running in solo mode, Brooklyn will install and configure the Chef client over SSH, just like it does most other kinds of entities.

Creating Blueprints from Chef

In a nutshell, a new Chef-based [entity](#) can be defined as a service by specifying `chef:cookbook_name` as the `service_type`, along with a collection of optional configuration. An illustrative example is below:

```
name: chef-mysql-sample
services:
- type: chef:mysql

brooklyn.config:
  cookbook_urls:
    # only needed for chef solo; URL can be local to brooklyn, or github, etc...
    mysql: https://github.com/opscode-cookbooks/mysql/archive/v4.0.12.tar.gz
    openssl: https://github.com/opscode-cookbooks/openssl/archive/v1.1.0.tar.gz
    build-essential: https://github.com/opscode-cookbooks/build-essential/archive/v1.4.4.tar.gz

  launch_run_list: [ "mysql::server" ]
  launch_attributes:
    mysql:
      # these attrs are required by the mysql cookbook under node['mysql']
      server_root_password: p4ssw0rd
      server_repl_password: p4ssw0rd
      server_debian_password: p4ssw0rd
      # many others are attrs are supported by the cookbook and can be passed here...

    # how to determine if the process is running and how to kill it
    # (supported options are `service_name` and `pid_file`; normally you should just pick one.
    # here we use the pid_file because the service_name varies, mysql on centos, mysqld on ubuntu!)
    #service_name: mysqld
    pid_file: /var/run/mysqld/mysqld.pid
```

This works without any installation: try it now, copying-and-pasting to the Brooklyn console. (Don't forget to add your preferred `location: some-cloud` to the spec.)

Notice, if you target `google-compute-engine` [location](#), you may need to specify `bind_address: 0.0.0.0` for the `mysql` cookbook, as described [here](#).

We'll now walk through the important constituent parts, and then proceed to describing things which can be done to simplify the deployment.

Cookbook Primary Name

The first thing to note is the type definition:

```
- type: chef:mysql
```

This indicates that the Chef [entity](#) should be used (`org.apache.brooklyn.entity.chef.ChefEntity`) to interpret and pass the configuration, and that it should be parameterised with a `brooklyn.chef.cookbook.primary.name` of `mysql`. This is the cookbook namespace used by default for determining what to install and run.

Importing Cookbooks

Next we specify which cookbooks are required and where they can be pulled from:

```
cookbook_urls:
  mysql: https://github.com/opscode-cookbooks/mysql/archive/v4.0.12.tar.gz
```

```

openssl: https://github.com/opscode-cookbooks/openssl/archive/v1.1.0.tar.gz
build-essential: https://github.com/opscode-cookbooks/build-essential/archive/v1.4.4.tar.gz

```

Here, specific versions are being downloaded from the canonical github repository. Any URL can be used, so long as it is resolvable on either the target machine or the Brooklyn server; this includes `file:` and `classpath:` URLs.

The archive can be ZIP or TAR or TGZ.

The structure of the archive must be that a single folder is off the root, and in that folder contains the usual Chef recipe and auxiliary files. For example, the archive might contain `mysql-master/recipes/server.rb`. Archives such as those above from github match this format.

The name of that folder does not matter, as often they contain version information. When deployed, these will be renamed to match the short name (the key in the `cookbooks_url` map, for instance `mysql` or `openssl`).

If Chef server is configured (see below), this section can be omitted.

Launch Run List and Attributes

The next part is to specify the Chef run list and attributes to store when launching the `entity`:

```

launch_run_list:
- mysql::server

launch_attributes:
  mysql:
    server_root_password: p4ssw0rd
    server_rep1_password: p4ssw0rd
    server_debian_password: p4ssw0rd

```

For the `launch_run_list`, you can use either the `YAML - recipe` syntax or the `JSON ["recipe"]` syntax.

The `launch_attributes` key takes a map which will be stored against the `node` object in Chef. Thus in this example, the parameter `node['mysql']['server_root_password']` required by the `mysql` `blueprint` is set as specified.

You can of course set many other attributes in this manner, in addition to those that are required!

Simple Monitoring

The final section determines how Brooklyn confirms that the service is up. Sophisticated solutions may install monitoring agents as part of the `launch_run_list`, with Brooklyn configured to read monitoring information to confirm the launch was successful. However for convenience, two common mechanisms are available out of the box:

```

#service_name: mysqld
pid_file: /var/run/mysqld/mysqld.pid

```

If `service_name` is supplied, Brooklyn will check the return code of the `status` command run against that service, ensuring it is 0. (Note that this is not universally reliable, although it is the same mechanism which Chef typically uses to test status when determining whether to start a service. Some services, e.g. `postgres`, will return 0 even if the service is not running.)

If a `pid_file` is supplied, Brooklyn will check whether a process with the PID specified in that file is running. This has been selected for `mysql` because it appears to be more portable: the service name varies among OS's: it is `mysqld` on CentOS but `mysql` on Ubuntu!

Writing Chef for Blueprints

Making it Simpler

The example we've just seen shows how existing Chef cookbooks can be used as the basis for entities. If you're *writing* the Chef recipes, there are a few simple techniques we've established with the Chef community which make blueprints literally as simple as:

```
- type: chef:mysql
brooklyn.config:
  mysql_password: p4ssw0rd
  pid_file: /var/run/mysqld/mysqld.pid
```

Some Basic Conventions

- **A start recipe:** The first step is to provide a `start` recipe in `recipes/start.rb`; if no `launch_run_list` is supplied, this is what will be invoked to launch the *entity*. It can be as simple as a one-line file:

```
include_recipe 'mysql::server'
```

- **Using `brooklyn.config`** : All the `brooklyn.config` is passed to Chef as node attributes in the `node['brooklyn']` `['config']` namespace. Thus if the required attributes in the `mysql` recipe are set to take a value set in `node['brooklyn']['config']['mysql_password']`, you can dispense with the `launch_attributes` section.

Using Chef Server

The examples so far have not required Chef Server, so they will work without any external Chef dependencies (besides the built-in install from <https://www.opscode.com/chef/install.sh> and the explicitly referenced cookbooks). If you use Chef Server, however, you'll want your managed nodes to be integrated with it. This is easy to set up, with a few options:

If you have `knife` set up in your shell environment, the Brooklyn Chef support will use it by default. If the recipes are installed in your Chef server, you can go ahead and remove the `cookbooks_url` section!

Use of `solo` or `knife` can be forced by setting the `chef_mode` flag (`brooklyn.chef.mode` config key) to either of those values. (It defaults to `autodetect`, which will use `knife` if it is on the path and satisfies sanity checks).

If you want to specify a different configuration, there are a number of config keys you can use:

- `brooklyn.chef.knife.executableFile` : this should be point to the knife binary to use
- `brooklyn.chef.knife.configFile` : this should point to the knife configuration to use
- `brooklyn.chef.knife.setupCommands` : an optional set of commands to run prior to invoking knife, for example to run `rvm` to get the right ruby version on the Brooklyn server

If you're interested in seeing the Chef REST API be supported directly (without knife), please let us know. We'd like to see this too, and we'll help you along the way!

Tips and Tricks

To help you on your way writing Chef blueprints, here are a handful of pointers particularly useful in this context:

- Configuration keys can be inherited from the top-level and accessed using `$brooklyn:entity('id').config('key_name')`. An example of this is shown in the `mysql-chef.yaml` sample recipe contained in the Brooklyn code base and [here](#) for convenience. Here, `password` is specified only once and then used for all the attributes required by the stock mysql cookbook.
- Github tarball downloads! You'll have noticed these in the example already, but they are so useful we thought we'd call them out again. Except when you're developing, we recommend using specific tagged versions rather than master.
- The usual machine `provisioning.properties` are supported with Chef blueprints, so you can set things like `minRam` and `osFamily`
- To see more configuration options, and understand the ones presented here in more detail, see the javadoc or the code for the class `chefconfig` in the Brooklyn code base.

Advanced Chef Integration

Adding Sensors and Effectors

Custom sensors and effectors can be added using an `entity.initializer` section in the [YAML blueprint](#).

One common pattern is to have sensors which extract information from Ohai. Another common pattern is to install a monitoring agent as part of the run list, configured to talk to a monitoring store, and then to add a [sensor](#) feed which reads data from that store.

On the [effector](#) side, you can add SSH-based effectors in the usual way. You can also describe additional chef converge targets following the pattern set down in `chefLifecycleEffectorTasks`, making use of conveniences in `chefSoloTasks` and `chefServerTasks`, or provide effectors which invoke network API's of the systems under management (for example to supply the common `executeScript` [effector](#) as on the standard `MySQLNode`).

Next Steps: Simplifying sensors and effectors, transferring files, and configuring ports

The Brooklyn-Chef integration is work in progress, with a few open issues we'd still like to add. Much of the thinking for this is set forth in the [Google document](#) indicated earlier. If you'd like to work with us to implement these, please let us know.

Reference

A general schema for the supported [YAML](#) is below:

```
- type: chef:cookbook_name
  brooklyn.config:
    cookbook_urls:
      cookbook_name: url://for/cookbook.tgz
      dependency1: url://for/dependency1.tgz
    launch_run_list: [ "cookbook_name::start" ]
    launch_attributes: # map of arguments to set in the chef node
    service_name: cookbook_service
    pid_file: /var/run/cookbook.pid
```

If you are interested in exploring the Java code for creating blueprints, start with the `TypedToyMySQLEntiyChef` class, which essentially does what this tutorial has shown; and then move on to the `DynamicToyMySQLEntiyChef` which starts to look at more sophisticated constructs. (Familiarity with BASH and basic Java blueprints may be useful at that stage.)

Salt in YAML Blueprints

This guide describes how Brooklyn entities can be created using the Salt infrastructure management tool (saltstack.com). At present Brooklyn provides basic support for Salt, operating in a 'masterless' mode. Comments on this support and suggestions for further development are welcome.

This guide assumes you are familiar with the basics of [creating YAML blueprints](#).

- [About Salt](#)
- [Creating Blueprints with Salt](#)

About Salt

What you need to know about Salt

Salt is designed to work in either what it calls a 'master/minion' or a 'masterless' topology.

In the former, the master server acts as a managing controller for any number of client nodes, called 'minions'. A salt daemon running on the minion connects back to the master server for its operation, and manages the software on the minion according to a specification of 'states' defined in Salt configuration files. In the latter, there is no master, and the salt daemon on the minion operates based on the Salt files on the minion node. This is the mode currently supported by Brooklyn.

A 'State' in Salt is a specification of a configuration of some aspect of a system, such as what packages are installed, what system services are running, what files exist, etc. Such states are described in "SLS" files (for SaLt State file). These files are typically written as templates using the "Jinja" templating engine. The actual SLS files for the minion are then created by processing the templates using configuration data provided via Salt's "Pillar" system.

Salt comes with built-in support for many software systems, and has a repository of pre-written Salt states known as 'formulas' on GitHub, at <https://github.com/saltstack-formulas>.

How Brooklyn interacts with Salt

Brooklyn provides a Salt **entity** type. An **entity** of this type can be specified in a **blueprint** in order to provision the node through Salt. The Salt **entity** will download Salt and install it on the node. The **entity** type supports the configuration of Salt formulas and Pillar data to download, and the configuration of what Salt states to apply. These are managed using Salt in 'masterless' mode, as described on the [Saltstack site](#), using the 'salt-call' functionality of Salt.

The Salt 'highstate' (the collection of states applied to the system) is exposed as a **sensor** on the **entity**. An **effector** is provided on the **entity** that supports general purpose Salt instructions.

Creating Blueprints with Salt

To write a [blueprint](#) to use Salt with Brooklyn it will help to have a degree of familiarity with Salt itself. In the sections below, when the Brooklyn configuration is described, the underlying Salt operation is also noted briefly, for clarity for readers who know Salt.

To manage a node with Salt, create a [blueprint](#) containing a service of type

`org.apache.brooklyn.entity.cm.salt.SaltEntity` and define the `formulas` and `start_states` For example:

```
name: Salt Example setting up Apache httpd
location: my-cloud
services:
- id: httpd-from-salt
  type: org.apache.brooklyn.entity.cm.salt.SaltEntity
  formulas:
  - https://github.com/saltstack-formulas/apache-formula/archive/master.tar.gz
  start_states:
  - apache
```

This example specifies that Brooklyn should use Salt to download the `apache-formula` from the Saltstack repository on Github. The apache formula contains the Apache web server with a simple "it worked" style index page. To start the [entity](#), Brooklyn will use Salt to apply the `apache` state, which will bring up the web server.

A typical usage of the Salt [entity](#) might be to include a formula from the Saltstack repository, such as `apache` above, and another formula created by the [blueprint](#) author, with additional states, such as web site content for the apache server.

Start States

The `start_states` configuration key defines the top level list of states that will be applied using Salt. These values are added to the Salt `top.sls` file and applied using `state.apply`. This configuration key is mandatory.

Stop States

The `stop_states` configuration key is used to specify states that should be applied when the 'stop' [effector](#) is invoked on the [entity](#). For example, the Saltstack `mysql` [formula](#) supplies a state `mysql.disabled` that will shut down the database server.

If the Saltstack formula does not supply a suitable stop state, the [blueprint](#) author can create a suitable state and include it in an additional formula to be supplied in the `formulas` section.

The `stop_states` configuration key is optional; if it is not provided, Brooklyn assumes that each state `s` in the `start_states` will have a matching `s.stop` state.

If any `s` does not have such a state, the stop [effector](#) will fail stopping processes. Note that on a machine created for this [entity](#), Brooklyn's default behaviour may be to proceed to destroy the VM, so stop states are not always needed unless there is a cleaner shutdown process or you are using long-running servers.

Restart States

For completeness, Brooklyn also provides a `restart_states` configuration key. These states are applied by the restart [effector](#), and [blueprint](#) authors may choose to provide custom states to implement restart if that is applicable for their application.

This key is again optional. If not supplied, Brooklyn will go through each of the states `s` in `start_states` looking for a matching `s.restart` state defined in the formulas. If all exist, these will be applied by the restart `effector`. If none exist, Brooklyn will invoke the `stop` and then `start` effectors -- so `restart` states are not required for Brooklyn to use Salt. (If some but not all have matching `restart` states, Brooklyn will fail the restart, on the assumption that the configuration is incomplete.)

Formulas

The `formulas` key provides the URLs for archives containing the Salt formulas that defined the required states. These archives are downloaded to the `/srv/formula` directory on the minion and added to the state filesystem roots configuration in Salt's minion config, so that their states are available for a `state.apply`.

Pillar Configuration

A typical Salt deployment will include both states (provided via Salt formulas) and configuration data, provided through Salt's "Pillar" component. Brooklyn provides configuration keys for the Salt `entity` to specify where to get the Pillar configuration data. For example:

```
name: Salt Example setting up MySQL with a Pillar
location: my-cloud
services:
- id: mysql-from-salt-with-my-pillar
  type: org.apache.brooklyn.entity.cm.salt.SaltEntity

formulas:
- https://github.com/saltstack-formulas/mysql-formula/archive/master.tar.gz
- http://myhost:8080/my-mysql-formula.tar.gz

start_states:
- mysql
stop_states:
- mysql.disabled

pillars:
- mysql
pillarUrls:
- http://myhost:8080/my-mysql-pillar.tar.gz
```

This `blueprint` contains the MySQL database, and includes a formula available from `myhost` which includes the schema information for the DB. The MySQL formula from Saltstack has extensive configurability through Salt Pillars. In the `blueprint` above, Brooklyn is instructed to apply the pillars defined in the `pillars` configuration key. (This will add these values to the Salt Pillars `top.sls` file.) The pillar data must be downloaded; for this, the `pillarUrls` key provides the address of an archive containing the Pillar data. The contents of the archive will be extracted and put in the `/srv/pillar` directory on the minion, in order to be available to Salt when applying the pillar. For example, the archive above can simply have the structure

```
pillar/
|
+- mysql/
  |
  +- init.sls
```

The `init.sls` contains the pillar configuration values, such as

```
# Manage databases
database:
```

```
- orders
schema:
  orders:
    load: True
    source: salt://mysql/files/orders.schema
```

Meanwhile the `my-mysql-formula.tar.gz` formula archive contains the schema:

```
my-mysql-formula/
|
+- mysql/
  |
  +- files/
    |
    +- orders.schema
```

Note that the `blueprint` above defines an `id` for the Salt `entity`. This id, if provided, is set as the minion id in the Salt configuration file. This is useful particularly for Pillar configuration, as, if there are more than one Salt managed nodes in the application, they can share a common Pillar file, with appropriate subdivisions of pillar data based on minion id.

Highstate Sensors

The Salt `entity` exposes the Salt "highstate" on the node via Brooklyn sensors. Firstly a single `sensor` `salt.states` contains a list of all the top level Salt state ID declarations in the highstate. For example, for the mysql case above, this might look like:

```
["mysql_additional_config", "mysql_config", "mysql_db_0", "mysql_db_0_load", "mysql_db_0_schema", "mysql_debconf",
"mysql_debconf_utils", "mysql_python", "mysql_user_frank_localhost", "mysql_user_frank_localhost_0",
"mysql_user_nopassuser_localhost", "mysqld"]
```

Then, for each ID and each Salt state function in that ID, a Brooklyn `sensor` is created, containing a map of the data from the highstate. For example, the `salt.state.mysqld.service.running` `sensor` would have a value like:

```
{"name":"mysql", "enable":true, "watch":[{"pkg":"mysqld"}, {"file":"mysql_config"}], "order":10005}
```

saltCall Effector

The Salt `entity` includes a general purpose Salt `effector`, `saltcall`, which permits execution of Salt commands via `salt-call --local`. It contains a single parameter, `spec`, which specifies the command to invoke. For example, invoking the `effector` with a `spec` value of `network.interfaces --out=yaml` would return a `YAML` formatted map of the network interfaces on the minion.

YAML Blueprint Advanced Example

By this point you should be familiar with the fundamental concepts behind both Apache Brooklyn and [YAML](#) blueprints. This section of the documentation is intended to show a complete, advanced example of a [YAML blueprint](#).

The intention is that this example is used to learn the more in-depth concepts, and also to serve as a reference when writing your own blueprints. This page will first explain what the example application is and how to run it, then it will spotlight interesting features.

Please note, there is now a much more up-to-date ELK [blueprint](#) that can be found [here](#). We've using an older version of this in the tutorial as it highlights some key Brooklyn concepts.

ELK Stack Example

This example demonstrates the deployment of an ELK Stack (Elasticsearch, Logstash and Kibana), using the provided [blueprint](#) to deploy, install, run and manage all three. Briefly, the component parts are:

- Elasticsearch: A clustered search engine
- Logstash: Collects, parses and stores logs. For our example it will store logs in Elasticsearch
- Kibana: A web front end to Elasticsearch

We also deploy a simple webserver whose logs will be collected.

- Tomcat 8: Web server whose logs will be stored in Elasticsearch by Logstash.

For more about the ELK stack, please see the documentation [here](#).

The Blueprints

There are four blueprints that make up this application. Each of them are used to add one or more catalog items to Brooklyn. You can find them below:

- [Elasticsearch](#)
- [Logstash](#)
- [Kibana](#)
- [ELK](#)

Running the example

First, add all four blueprints to the Brooklyn Catalog. This can be done by clicking the 'Catalog' tab, clicking the '+' symbol and pasting the [YAML](#). Once this is done, click the 'Application' tab, then the '+' button to bring up the add application wizard. A new Catalog application will be available called 'ELK Stack'. Using the add application wizard, you should be able to deploy an ELK stack to a [location](#) of your choosing. Alternatively use the `br` Brooklyn command line tool and add the files with `br catalog add`.

Exploring the example

After the application has been deployed, you can ensure it is working as expected by checking the following:

- There is a Kibana [sensor](#) called `main.uri`, the value of which points to the Kibana front end. You can explore this

front end, and observe the logs stored in Elasticsearch. Many Brooklyn applications have a `main.uri` set to point you in the right direction.

- You can also use the Elasticsearch REST API to explore further. The Elasticsearch Cluster `entity` has a `urls.http.list` `sensor`. Using a host:port from that list you will be able to access the REST API. The following URL will give you the state of the cluster `http://<host:port>/_cluster/health?pretty=true`. As you can see the `number_of_nodes` is currently 2, indicating that the Elasticsearch nodes are communicating with each other.

Interesting Feature Spotlight

We will mainly focus on the Elasticsearch `blueprint`, and will be clear when another `blueprint` is being discussed. This `blueprint` describes a cluster of Elasticsearch nodes.

Provisioning Properties

Our Elasticsearch `blueprint` has a few requirements of the `location` in which it is run. Firstly, it must be run on an Ubuntu machine as the example has been written specifically for this OS. Secondly, two ports must be opened to ensure that the entities can be accessed from the outside world. Both of these requirements are configured via `provisioning.properties` as follows:

```
brooklyn.config:
  elasticsearch.http.port: 9220
  elasticsearch.tcp.port: 9330
  provisioning.properties:
    osFamily: ubuntu
    inboundPorts:
      - $brooklyn:config("elasticsearch.http.port")
      - $brooklyn:config("elasticsearch.tcp.port")
```

VanillaSoftwareProcess

When composing a `YAML blueprint`, the `VanillaSoftwareProcess` is a very useful `entity` to be aware of. A `VanillaSoftwareProcess` will instruct Brooklyn to provision an instance, and run a series of shell commands to setup, run, monitor and teardown your program. The commands are specified as configuration on the `VanillaSoftwareProcess` and there are several available. We will spotlight a few now. To simplify this `blueprint`, we have specified `ubuntu` only installs so that our commands can be tailored to this system (e.g. use `apt-get` rather than `yum`).

Customize Command

The `Customize Command` is run after the application has been installed but before it is run. It is the perfect place to create and amend config files. Please refer to the following section of the Elasticsearch `blueprint`:

```
customize.command: |
  sudo rm -fr sudo tee /etc/elasticsearch/elasticsearch.yml
  echo discovery.zen.ping.multicast.enabled: false | sudo tee -a /etc/elasticsearch/elasticsearch.yml
  echo discovery.zen.ping.unicast.enabled: true | sudo tee -a /etc/elasticsearch/elasticsearch.yml
  echo discovery.zen.ping.unicast.hosts: ${URLS_WITH_BRACKETS} | sudo tee -a /etc/elasticsearch/elasticsearch.y
ml
  echo http.port: ${ES_HTTP_PORT} | sudo tee -a /etc/elasticsearch/elasticsearch.yml
  echo transport.tcp.port: ${ES_TCP_PORT} | sudo tee -a /etc/elasticsearch/elasticsearch.yml
  echo network.host: ${IP_ADDRESS} | sudo tee -a /etc/elasticsearch/elasticsearch.yml
```

The purpose of this section is to create a `YAML` file with all of the required configuration. We use the `YAML` literal style `|` indicator to write a multi line command. We start our series of commands by using the `rm` command to remove the previous config file. We then use `echo` and `tee` to create the new config file and insert the config. Part of the

configuration is a list of all hosts that is set on the parent `entity`- this is done by using a combination of the `component` and `attributeWhenReady` DSL commands. More on how this is generated later.

Check running

After an app is installed and run, this command is scheduled to run regularly and used to populate the `service.isUp` `sensor`. If this command is not specified, or returns an exit code of anything other than zero, then Brooklyn will assume that your `entity` has failed and will display the fire status symbol. Please refer to the following section of the Elasticsearch blueprint:

```
checkRunning.command: sudo systemctl status kibana.service
```

There are many different ways to implement this command. For this example, we are simply using the `systemctl` status of the appropriate service.

Enrichers

Elasticsearch URLs

To ensure that all Elasticsearch nodes can communicate with each other they need to be configured with the TCP URL of all other nodes. Similarly, the Logstash instances need to be configured with all the HTTP URLs of the Elasticsearch nodes. The mechanism for doing this is the same, and involves using Transformers, Aggregators and Joiners, as follows:

```
brooklyn.enrichers:
- type: org.apache.brooklyn.enricher.stock.Transformer
  brooklyn.config:
    enricher.sourceSensor: $brooklyn:sensor("host.subnet.address")
    enricher.targetSensor: $brooklyn:sensor("url.tcp")
    enricher.targetValue: $brooklyn:formatString("%s:%s", $brooklyn:attributeWhenReady("host.subnet.address"),
, $brooklyn:config("elasticsearch.tcp.port"))
```

In this example, we take the `host.subnet.address` and append the TCP port, outputting the result as `url.tcp`. After this has been done, we now need to collect all the URLs into a list in the Cluster `entity`, as follows:

```
brooklyn.enrichers:
- type: org.apache.brooklyn.enricher.stock.Aggregator
  brooklyn.config:
    enricher.sourceSensor: $brooklyn:sensor("url.tcp")
    enricher.targetSensor: $brooklyn:sensor("urls.tcp.list")
    enricher.aggregating.fromMembers: true
```

In the preceding example, we aggregated all of the TCP URLs generated in the early example. These are then stored in a `sensor` called `urls.tcp.list`. This list is then joined together into one long string:

```
- type: org.apache.brooklyn.enricher.stock.Joiner
  brooklyn.config:
    enricher.sourceSensor: $brooklyn:sensor("urls.tcp.list")
    enricher.targetSensor: $brooklyn:sensor("urls.tcp.string")
    uniqueTag: urls.quoted.string
```

Finally, the string has brackets added to the start and end:

```
- type: org.apache.brooklyn.enricher.stock.Transformer
  brooklyn.config:
    enricher.sourceSensor: $brooklyn:sensor("urls.tcp.string")
    enricher.targetSensor: $brooklyn:sensor("urls.tcp.withBrackets")
    enricher.targetValue: $brooklyn:formatString("[%s]", $brooklyn:attributeWhenReady("urls.tcp.string"))
```

The resulting `sensor` will be called `urls.top.withBrackets` and will be used by all Elasticsearch nodes during setup.

Kibana URL

Kibana also needs to be configured such that it can access the Elasticsearch cluster. However, Kibana can only be configured to point at one Elasticsearch instance. To enable this, we use another `enricher` in the cluster to select the first URL from the list, as follows:

```
- type: org.apache.brooklyn.enricher.stock.Aggregator
  brooklyn.config:
    enricher.sourceSensor: $brooklyn:sensor("host.subnet.address")
    enricher.targetSensor: $brooklyn:sensor("host.address.first")
    enricher.aggregating.fromMembers: true
    enricher.transformation:
      $brooklyn:object:
        type: "org.apache.brooklyn.util.collections.CollectionFunctionals$FirstElementFunction"
```

Similar to the above Aggregator, this Aggregator collects all the URLs from the members of the cluster. However, this Aggregator specifies a transformation. In this instance a transformation is a Java class that implements a Guava Function `<? super Collection<?>, ?>`, i.e. a function that takes in a collection and returns something. In this case we specify the `FirstElementFunction` from the `CollectionFunctionals` to ensure that we only get the first member of the URL list.

Latches

In the ELK `blueprint`, there is a good example of a latch. Latches are used to force an `entity` to wait until certain conditions are met before continuing. For example:

```
- type: kibana-standalone
  id: kibana
  name: Kibana Server
  latch.customize: $brooklyn:component("es").attributeWhenReady("service.isUp")
```

This latch is used to stop Kibana customizing until the Elasticsearch cluster is up. We do this to ensure that the URL sensors have been setup, so that they can be passed into Kibana during the customization phase.

Latches can also be used to control how many entities can execute the same step at any given moment. When a latch is given the value of a `MaxConcurrencySensor` it will unblock execution only when there are available "slots" to execute (think of it as a semaphore). For example to let a single `entity` execute the launch step of the start `effector`:

```
services:
- type: cluster

  brooklyn.initializers:
  - type: org.apache.brooklyn.core.sensor.MaxConcurrencySensor
    brooklyn.config:
      name: single-executor
      latch.concurrency.max: 1

  brooklyn.config:
    initialSize: 10
    memberSpec:
      $brooklyn:entitySpec:
        type: vanilla-bash-server
        brooklyn.config:
          launch.command: sleep 2
          checkRunning.command: true
          latch.launch: $brooklyn:parent().attributeWhenReady("single-executor")
```

It's important to note that the above setup is not reentrant. This means that users should be careful to avoid deadlocks. For example having a start and launch latches against the `single-executor` from above. The launch latch will block forever since the start latch already would've acquired the free slot.

Child entities

The ELK `blueprint` also contains a good example of a child `entity`.

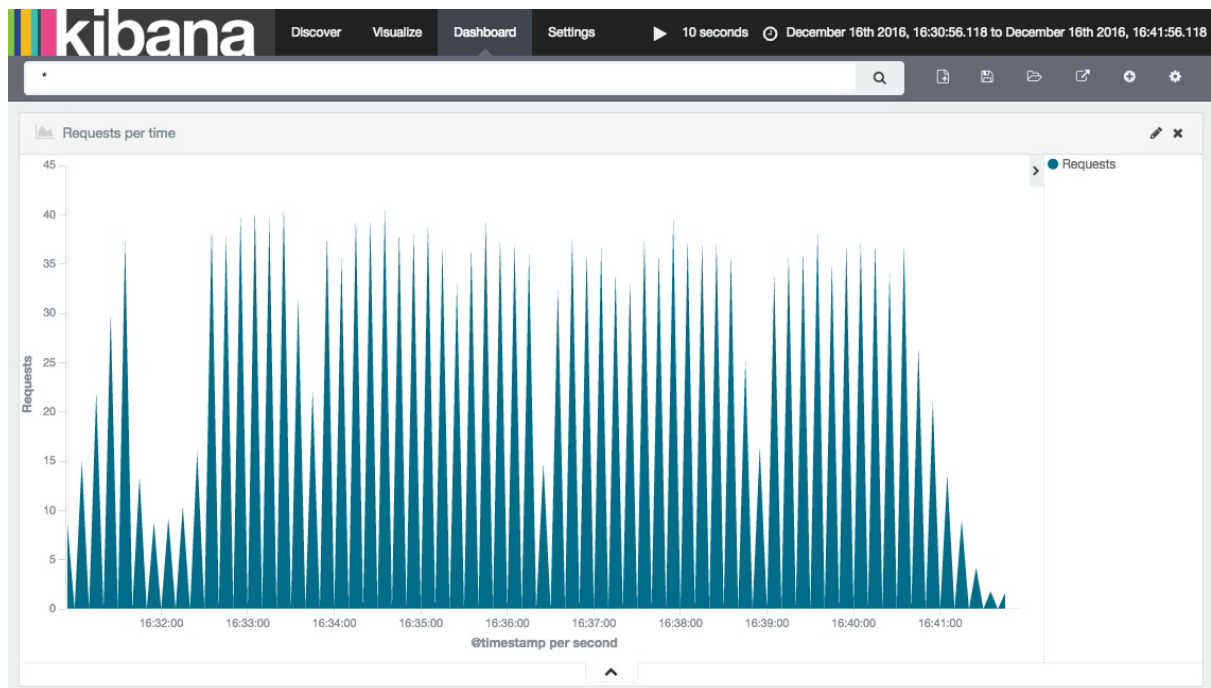
```
- type: org.apache.brooklyn.entity.webapp.tomcat.Tomcat8Server
  brooklyn.config:
    children.startable.mode: background_late
    ...
  brooklyn.children:
    - type: logstash-child
```

In this example, a `logstash-child` is started as a child of the parent Tomcat server. The Tomcat server needs to be configured with a `children.startable.mode` to inform Brooklyn when to bring up the child. In this case we have selected `background` so that the child is disassociated from the parent `entity`, and `late` to specify that the parent `entity` should start before we start the child.

The example also shows how to configure Logstash inputs and filters, if necessary, for a particular application, in this case Tomcat.

```
- type: logstash-child
  name: Logstash
  brooklyn.config:
    logstash.elasticsearch.hosts: $brooklyn:entity("es").attributeWhenReady("urls.http.withBrackets")
    logstash.config.input:
      $brooklyn:formatString:
        - |
          input {
            file {
              path => "%s/logs/localhost_access_log.*"
              start_position => "beginning"
            }
          }
        - $brooklyn:entity("tomcat").attributeWhenReady("run.dir")
    logstash.config.filter: |
      filter {
        grok {
          match => { "message" => "%{COMBINEDAPACHELOG}" }
        }
        date {
          match => [ "timestamp" , "dd/MMM/yyyy:HH:mm:ss Z" ]
        }
      }
  }
```

Configuring an appropriate visualisation on the Kibana server (access it via the URL on the summary tab for that `entity`) allows a dashboard to be created such as



Blueprinting Tips

YAML Recommended

The recommended way to write a [blueprint](#) is as a [YAML](#) file. This is true both for building an application out of existing blueprints, and for building new integrations.

The use of Java is reserved for those use-cases where the provisioning or configuration logic is very complicated.

Be Familiar with Brooklyn

Be familiar with the stock entities available in Brooklyn. For example, prove you understand the concepts by making a deployment of a cluster of Tomcat servers before you begin writing your own blueprints.

Ask For Help

Ask for help early. The community is keen to help, and also keen to find out what people find hard and how people use the product. Such feedback is invaluable for improving future versions.

Faster Dev-Test

Writing a [blueprint](#) is most efficient and simple when testing is fast, and when testing is done incrementally as features of the [blueprint](#) are written.

The slowest stages of deploying a [blueprint](#) are usually VM provisioning and downloading/installing of artifacts (e.g. RPMs, zip files, etc).

Options for speeding up provisioning include those below.

Deploying to Bring Your Own Nodes (BYON)

A [BYON location](#) can be defined, which avoids the time required to provision VMs. This is fast, but has the downside that artifacts installed during a previous run can interfere with subsequent runs.

A variant of this is to [use Vagrant](#) (e.g. with VirtualBox) to create VMs on your local machine, and to use these as the target for a [BYON location](#).

These VMs should mirror the target environment as much as possible.

Deploying to the "localhost" location

This is fast and simple, but has some obvious downsides:

- Artifacts are installed directly on your desktop/server.
- The artifacts installed during previous runs can interfere with subsequent runs.
- Some entities require `sudo` rights, which must be granted to the user running Brooklyn.

Deploying to Clocker

Docker containers provide a convenient way to test blueprints (and also to run blueprints in production!).

The [Clocker project](#) allows the simple setup of Docker Engine(s), and for Docker containers to be used instead of VMs. For testing, this allows each run to start from a fresh container (i.e. no install artifacts from previous runs), while taking advantage of the speed of starting containers.

Local Repository of Install Artifacts

To avoid re-downloading install artifacts on every run, these can be saved to `~/.brooklyn/repository/`. The file structure is a sub-directory with the [entity](#)'s simple name, then a sub-directory with the version number, and then the files to be downloaded. For example, `~/.brooklyn/repository/TomcatServer/7.0.56/apache-tomcat-7.0.56.tar.gz`.

If possible, synchronise this directory with your test VMs.

Re-install on BYON

If using BYON or localhost, the install artifacts will by default be installed to a directory like `/tmp/brooklyn-myname/installs/`. If install completed successfully, then the install stage will be subsequently skipped (a marker file being used to indicate completion). To re-test the install phase, delete the install directory (e.g. delete `/tmp/brooklyn-myname/installs/TomcatServer_7.0.56/`).

Where installation used something like `apt-get install` or `yum install`, then re-testing the install phase will require uninstalling these artifacts manually.

Monitoring and Managing Applications

Think about what it really means for an application to be running. The out-of-the-box default for a software process is the lowest common denominator: that the process is still running. Consider checking that the app responds over HTTP etc.

If you have control of the app code, then consider adding an explicit health check URL that does more than basic connectivity tests. For example, can it reach the database, message broker, and other components that it will need for different operations.

Writing Composite Blueprints

Write everything in discrete chunks that can be composed into larger pieces. Do not write a single mega-blueprint. For example, ensure each component is added to the catalog independently, along with a [blueprint](#) for the composite app.

Experiment with lots of small blueprints to test independent areas before combining them into the real thing.

Writing Entity Tests

Use the [test framework](#) to write test cases. This will make automated (regression) testing easier, and will allow others to easily confirm that the [entity](#) works in their environment.

If using Maven/Gradle then use the [Brooklyn Maven plugin](#) to test blueprints at build time.

Custom Entity Development

If writing a custom integration, the following recommendations may be useful:

- Always be comfortable installing and running the process yourself before attempting to automate it.
- For the software to be installed, use its Installation and Admin guides to ensure best practices are being followed. Use blogs and advice from experts, when available.
- Where there is a choice of installation approaches, use the approach that is most appropriate for production use-cases (even if this is harder to test on localhost). For example, prefer the use of RPMs versus unpacking zip files, and prefer the use of services versus invoking a `bin/start` script.
- Ensure every step is scriptable (e.g. manual install does not involve using a text editor to modify configuration files, or clicking on things in a web-console).
- Write scripts (or Chef recipes, or Puppet manifests, etc), and test these by executing manually. Only once these work in isolation, add them to the [entity blueprint](#).
- Externalise the configuration where appropriate. For example, if there is a configuration file then include a config key for the URL of the configuration file to use. Consider using FreeMarker templating for such configuration files.
- Focus on a single OS distro/version first, and clearly document these assumptions.
- Breakdown the integration into separate components, where possible (and thus develop/test them separately). For example, if deploying a MongoDB cluster then first focus on single-node MongoDB, and then make that configurable and composable for a cluster.
- Where appropriate, share the new [entity](#) with the Brooklyn community so that it can be reviewed, tested and improved by others in the community!

Cloud Portability

You get a lot of support out-of-the-box for deploying blueprints to different clouds. The points below may also be of help:

- Test (and regression test) on each target cloud.
- Be aware that images on different clouds can be very different. For example, two CentOS 6.6 VMs might have different pre-installed libraries, different default iptables or SELinux settings, different repos, different sudo configuration, etc.
- Different clouds handle private and public IPs differently. One must be careful about which address to advertise to for use by other entities.
- VMs on some clouds may not have a well-configured hostname (e.g. `ping $(hostname)` can fail).
- VMs in different clouds have a different number of NICs. This is important when choosing whether to listen on 0.0.0.0 or on a specific NIC.

Investigating Errors

ALWAYS keep logs when there is an error.

See the [Troubleshooting](#) guide for more information.

YAML Blueprint Reference

Root Elements

- `name` : human readable names
- `services` : a list of `ServiceSpecification` elements
- `location` (or `locations` taking a list): a `LocationSpecification` element as a string or a map

Service Specification Elements

Within the `services` block, a list of maps should be supplied, with each map defining a `ServiceSpecification`. Each `ServiceSpecification` should declare the service `type` (synonyms `serviceType` and `service_type`), indicating what type of service is being specified there. The following formats are supported for defining types:

- `com.acme.brooklyn.package.JavaEntityClass`
- `java:com.acme.brooklyn.package.JavaEntityClass`
- `java-entity-class` (where this has been added to the `catalog`)

A reference of some of the common service `type` instances used is included in a section below.

Within the `ServiceSpecification`, other key-value pairs can be supplied to customize the `entity` being defined, with these being the most common:

- `id` : an ID string, used to refer to this service
- `location` (or `locations`): as defined in the root element
- `brooklyn.config` : configuration key-value pairs passed to the service `entity` being created
- `brooklyn.children` : a list of `ServiceSpecifications` which will be configured as children of this `entity`
- `brooklyn.policies` : a list of policies, each as a map described with their `type` and their `brooklyn.config` as keys
- `brooklyn.enrichers` : a list of enrichers, each as a map described with their `type` and their `brooklyn.config` as keys; see the keys declared on individual enrichers; also see [this enricher example](#) for a detailed and commented illustration
- `brooklyn.initializers` : a list of `EntityInitializer` instances to be constructed and run against the `entity`, each as a map described with their `type` and their `brooklyn.config` as keys. An `EntityInitialiizer` can perform arbitrary customization to an `entity` whilst it is being constructed, such as adding dynamic sensors and effectors. These classes must expose a public constructor taking a single `Map` where the `brooklyn.config` is passed in. Some common initializers are:
 - `org.apache.brooklyn.core effector.ssh.SshCommandEffector` : takes a `name` and `command`, and optionally a map of named `parameters` to their `description` and `defaultValue`, to define an `effector` with the given name implemented by the given SSH command (on an `entity` which as an ssh-able machine)
 - `org.apache.brooklyn.core.sensor.ssh.SshCommandSensor` : takes a `name` and `command`, and optionally a `period`, to create a `sensor` feed which populates the `sensor` with the given name by running the given command (on an `entity` which as an ssh-able machine)

- `org.apache.brooklyn.core.sensor.windows.WinRmCommandSensor` : For a command supplied via WinRm. Takes a `name` , `command` , and optionally a `period` and `executionDir` , to create a `sensor` feed which populates the `sensor` with the given name by running the given command (on an `entity` which has a winrm-able machine).
`"~"` will use the default execution directory for the WinRm session which is usually `%USERPROFILE%`
- `brooklyn.parameters` : documents a list of typed parameters the `entity` accepts. If none are specified the config keys declared in the `entity`'s class are used (including the information from the `@CatalogConfig` annotation). The items have the following properties:
 - `name` (required): identifier by which to reference the parameter when setting or retrieving its value
 - `label` : a value to present to the user, same as `name` if empty
 - `description` : short text describing the parameter behaviour/usage, presented to the user
 - `type` : the type of the parameter, one of `string` , `integer` , `long` , `float` , `double` , `timestamp` , `duration` , `port` , or a fully qualified Java type name; the default is `string` ; obvious coercion is supported so `timestamp` accepts most common ISO date formats, `duration` accepts `5m` , and `port` accepts `8080+`
 - `default` : a default value; this will be coerced to the declared `type`
 - `pinned` : mark the parameter as pinned (always displayed) for the UI. The default is `true`
 - `constraints` : a list of constraints the parameter should meet; for details, see [Entity Configuration](#).

A shorthand notation is also supported where just the name of the parameter is supplied as an item in the list, with the other values being unset or the default. See `displayName` in the following example for an illustration of this:

```
brooklyn.parameters:
# user.age parameter is required, pinned and fully specified
- name: user.age
  type: integer
  label: Age
  description: the age of the user
  pinned: true
  constraints:
  - required
# user.name is optional, is not pinned and has a default
- name: user.name
  default: You
  pinned: false
# shorthand notation: displayName will be an optional config of type string with no default
- displayName
```

Entities, policies, and initializers may accept additional key-value pairs, usually documented in their documentation (e.g. javadoc), or in the case of Java often as static fields in the underlying Java class. Often there are config keys or flags (indicated by `@SetFromFlag`) declared on the class; these declared flags and config keys may be passed in at the root of the `ServiceSpecification` or in `brooklyn.config` . (Undeclared config is only accepted in the `brooklyn.config` map.) Referencing the parameters from within java classes is identical to using config keys. In `yaml` it's usually referenced using `$brooklyn:scopeRoot().config("displayName")` . See below for more details on scopes.

- `brooklyn.tags` : documents a list of tag objects which should be assigned to the `entity`.

Location Specification Elements

In brief, `location` specs are supplied as follows, either for the entire application (at the root) or for a specific

`ServiceSpecification` :

```
location:
  jclouds:aws-ec2:
    region: us-east-1
    identity: AKA_YOUR_ACCESS_KEY_ID
    credential: <access-key-hex-digits>
```

Or in many cases it can be in-lined:

```
location: localhost
location: named:my_openstack
location: aws-ec2:us-west-1
```

For the first immediately, you'll need password-less ssh access to localhost. For the second, you'll need to define a named `location` in `brooklyn.properties`, using `brooklyn.location.named.my_openstack...` properties. For the third, you'll need to have the identity and credentials defined in `brooklyn.properties`, using `brooklyn.location.jclouds.aws-ec2...` properties.

If specifying multiple locations, e.g. for a fabric:

```
locations:
- localhost
- named:my_openstack
- aws-ec2:us-east-2 # if credentials defined in `brooklyn.properties`
- jclouds:aws-ec2:
  region: us-east-1
  identity: AKA_YOUR_ACCESS_KEY_ID
  credential: <access-key-hex-digits>
```

If you have pre-existing nodes, you can use the `byon` provider, either in this format:

```
location:
  byon:
    user: root
    privateKeyFile: ~/.ssh/key.pem
    hosts:
      - 81.95.144.58
      - 81.95.144.59
      - brooklyn@159.253.144.139
      - brooklyn@159.253.144.140
```

or:

```
location:
  byon:
    user: root
    privateKeyFile: ~/.ssh/key.pem
    hosts: "{81.95.144.{58,59},brooklyn@159.253.144.{139-140}}"
```

You cannot use glob expansions with the list notation, nor can you specify per-host information apart from user within a single `byon` declaration. However you can combine locations using `multi`:

```
location:
  multi:
    targets:
      - byon:
          user: root
          privateKeyFile: ~/.ssh/key.pem
          hosts:
            - 81.95.144.58
            - 81.95.144.59
      - byon:
          privateKeyFile: ~/.ssh/brooklyn_key.pem
          hosts: brooklyn@159.253.144{139-140}
```

DSL Commands

Dependency injection other powerful references and types can be built up within the [YAML](#) using the concise DSL defined here:

- `$brooklyn:attributeWhenReady("sensor")` will store a future which will be blocked when it is accessed, until the given `sensor` from this `entity` "truthy" (i.e. non-trivial, non-empty, non-zero) value (see below on `component` for looking up values on other sensors)
- `$brooklyn:config("key")` will insert the value set against the given key at this `entity` (or nearest ancestor); can be used to supply config at the root which is used in multiple places in the plan
- `$brooklyn:sensor("sensor.name")` returns the given `sensor` on the current `entity` if found, or an untyped (Object) `sensor`; `$brooklyn:sensor("com.acme.brooklyn.ContainingEntityClass", "sensor.name")` returns the strongly typed `sensor` defined in the given class
- `$brooklyn:entity("ID")` refers to a Brooklyn `entity` with the given ID; you can then access the following subfields, using the same syntax as defined above but with a different reference `entity`, e.g.
`$brooklyn:entity("ID").attributeWhenReady("sensor") :`
 - `.attributeWhenReady("sensor")`
 - `.config("key")`
 - `.sensor("sensor.name")`
- `$brooklyn:component("scope", "ID")` is also supported, to limit scope to any of
 - `global` : looks for the `ID` anywhere in the plan
 - `child` : looks for the `ID` anywhere in the child only
 - `descendant` : looks for the `ID` anywhere in children or their descendants
 - `sibling` : looks for the `ID` anywhere among children of the parent `entity`
 - `parent` : returns the parent `entity` (ignores the `ID`)
 - `this` : returns this `entity` (ignores the `ID`)
- `$brooklyn:root()` will return the topmost `entity` (the application)
- `$brooklyn:scopeRoot()` will return the root `entity` in the current plan scope. For catalog items it's the topmost `entity` in the plan, for application plans it is the same as `$brooklyn:root()` .
- `$brooklyn:formatString("pattern e.g. %s %s", "field 1", "field 2")` returns a future which creates the formatted string with the given parameters, where parameters may be strings or other tasks such as `attributeWhenReady`
- `$brooklyn:urlEncode("val")` returns a future which creates a string with the characters escaped so it is a valid part of a URL. The parameter can be a string or another task. For example,
`$brooklyn:urlEncode($brooklyn:config(\ "mykey\ "))` . It uses "www-form-urlencoded" for the encoding, which is appropriate for query parameters but not for some other parts of the URL (e.g. space is encoded as '+').
- `$brooklyn:literal("string")` returns the given string as a literal (suppressing any `$brooklyn:` expansion)
- `$brooklyn:object(Map)` creates an object, using keys `type` to define the java type, and either `object.fields` or `brooklyn.config` to supply bean/constructor/flags to create an instance
- `$brooklyn:entitySpec(Map)` returns a new `ServiceSpecification` as defined by the given `Map` , but as an `EntitySpec` suitable for setting as the value of `ConfigKey<EntitySpec>` config items (such as `dynamiccluster.memberspec` in `DynamicCluster`)

Parameters above can be supplied either as strings or as lists and maps in [YAML](#), and the `$brooklyn:` syntax can be used within those parameters.

Some Powerful YAML Entities

All entities support configuration via [YAML](#), but these entities in particular have been designed for general purpose use from [YAML](#). Consult the Javadoc for these elements for more information:

- **Vanilla Software** in `VanillaSoftwareProcess` : makes it very easy to build entities which use `bash` commands to

install and the PID to stop and restart

- **Chef** in `chefSoftwareProcess` : makes it easy to use Chef cookbooks to build entities, either with recipes following conventions or with configuration in the `serviceSpecification` to use arbitrary recipes
- `dynamicCluster` : provides resizable clusters given a `dynamicCluster.memberspec` set with `$brooklyn.entitySpec(Map)` as described above
- `dynamicFabric` : provides a set of homogeneous instances started in different locations, with an `effector` to `addLocation` , i.e. add a new instance in a given `location`, at runtime

Deploying Blueprint

Locations are the environments to which Brooklyn deploys applications. Most commonly these are cloud services such as AWS, GCE, and IBM Softlayer. Brooklyn also supports deploying to a pre-provisioned network or to localhost (primarily useful for testing blueprints).

See also:

- The [Locations yaml guide](#)
- Use within an [entity](#) of the configuration option [provisioning.properties](#)
- How to add [location](#) definitions to the [Catalog](#); and
- How to use [Externalized Configuration](#).

The requirements for how a provisioned machine should behave will depend on the entites subsequently deployed there.

Below are a set of common assumptions, made by many [entity](#) implementations, which could cause subsequent errors if they do not hold. These relate to the machine's configuration, rather than additional networking or security that a given Cloud might offer.

Also see the [Troubleshooting](#) docs.

Remote Access

SSH or WinRM Access

Many entities require ssh'ing (or using WinRM for Windows), to install and configure the software.

An example of disabling all ssh'ing is shown below:

```
location:
  aws-ec2:us-east-1:
    identity: XXXXXXXX
    credential: XXXXXXXX
    waitForSshable: false
    pollForFirstReachableAddress: false
services:
- type: org.apache.brooklyn.entity.software.base.EmptySoftwareProcess
  brooklyn.config:
    onbox.base.dir.skipResolution: true
    sshMonitoring.enabled: false
```

Parsing SSH stdout: No Extra Lines

For entities that execute ssh commands, these sometimes parse the resulting stdout.

It is strongly recommended that VMs are configured so that no additional stdout is written when executing remote ssh (or WinRM) commands. Such stdout risks interfering with the response parsing in some blueprints.

For example, if configuring the VM to write out "Last login" information, this should be done for only "interactive" shells (see [Stackoverflow](#) for more details).

Passwordless Sudo

Does passwordless sudo work?

Try executing:

```
sudo whoami
```

See [Passwordless Sudo](#).

Advertised Addresses

Hostname Resolves Locally

Does the hostname known at the box resolve at the box?

Try executing:

```
ping $(hostname)
```

if not, consider setting `generate.hostname: true` in the [location](#) config, for jclouds-based locations.

IP Resolves Locally

For the IP address advertised in Brooklyn using the [sensor](#) `host.addresses.private` (or `host.subnet.address`), can the machine reach that IP?

Get the [sensor](#) value, and then try executing:

```
ping ${PRIVATE_IP}
```

Is there a public IP (advertised using the [sensor](#) `host.addresses.public`, or `host.address`), and can the machine reach it?

Get the [sensor](#) value, and then try executing:

```
ping ${PUBLIC_IP}
```

Networking

Public Internet Access

Can the machine reach the public internet, and does DNS resolve?

Try executing:

```
ping www.example.org
```

Machine's Hostname in DNS

Is the machine hostname well-known? If one does a DNS lookup, e.g. from the Brooklyn server, does it resolve and does it return the expected IP (e.g. the same IP as the `host.addresses.public` `sensor`)? Try using the hostname that the machine reports when you execute `hostname`.

Many blueprints do not require this, instead using IP addresses directly. Some blueprints may include registration with an appropriate DNS server. Some clouds do this automatically.

Reachability

When provisioning two machines, can these two machines reach each other on the expected IP(s) and hostname(s)?

Try using `ping` from one machine to another using the public or subnet ip or hostname. However, note that `ping` requires access over ICMP, which may be disabled. Alternatively, try connecting to a specific TCP port using `telnet <address> <port>`.

Firewalls

What firewall(s) are running on the machine, and are the required ports open? On linux, check things like `iptables`, `firewalld`, `ufw` or other commercial firewalls. On Windows, check the settings of the [Windows Firewall](#).

Consider using `openIptables: true`, or even `stopIptables: true`.

Sufficient Entropy for /dev/random

Is there sufficient entropy on the machine, for `/dev/random` to respond quickly?

Try executing:

```
{ cat /dev/random > /tmp/x & } ; sleep 10 ; kill %1 ; { cat /dev/random > /tmp/x & } ; sleep 1 ; kill %1 ; wc /tmp/x | awk '{print $3}'
```

The result should be more than 1M.

If not, consider setting `installDevUrandom: true` for jclouds-based locations.

See instructions to [Increase Entropy](#).

File System

Permissions of /tmp

Is `/tmp` writable?

Try executing:

```
touch /tmp/amp-test-file ; rm /tmp/amp-test-file
```

Are files in `/tmp` executable (e.g. some places it has been mounted NO_EXECUTE)?

Try executing:

```
echo date > /tmp/brooklyn-test.sh && chmod +x /tmp/brooklyn-test.sh && /tmp/brooklyn-test.sh && rm /tmp/brooklyn-test.sh
```

```
n-test.sh
```

Clouds

For most cloud provisioning tasks, Brooklyn uses [Apache jclouds](#). The identifiers for some of the most commonly used jclouds-supported clouds are (or [see the full list](#)):

- `jclouds:aws-ec2:<region>` : Amazon EC2, where `<region>` might be `us-east-1` or `eu-west-1` (or omitted)
- `jclouds:softlayer:<region>` : IBM Softlayer, where `<region>` might be `dal05` or `ams01` (or omitted)
- `jclouds:google-compute-engine` : Google Compute Engine
- `jclouds:openstack-nova:<endpoint>` : OpenStack, where `<endpoint>` is the access URL (required)
- `jclouds:cloudstack:<endpoint>` : Apache CloudStack, where `<endpoint>` is the access URL (required)

For any of these, of course, Brooklyn needs to be configured with an `identity` and a `credential` :

```
location:
  jclouds:aws-ec2:
    identity: ABCDEFGHIJKLMNOPQRST
    credential: s3cr3tsq1rr3ls3cr3tsq1rr3ls3cr3tsq1rr3l
```

The above [YAML](#) can be embedded directly in blueprints, either at the root or on individual services. If you prefer to keep the credentials separate, you can instead store them as a [catalog entry](#) or set them in `brooklyn.properties` in the `jclouds.<provider>` namespace:

```
brooklyn.location.jclouds.aws-ec2.identity=ABCDEFGHIJKLMNOPQRST
brooklyn.location.jclouds.aws-ec2.credential=s3cr3tsq1rr3ls3cr3tsq1rr3ls3cr3tsq1rr3l
```

And in this case you can reference the [location](#) in [YAML](#) with `location: jclouds:aws-ec2` .

Alternatively, you can use the [location](#) wizard tool available within the web console to create any cloud [location](#) supported by [Apache jclouds](#). This [location](#) will be saved as a [catalog entry](#) for easy reusability.

Brooklyn irons out many of the differences between clouds so that blueprints run similarly in a wide range of locations, including setting up access and configuring images and machine specs. The configuration options are described in more detail below.

In some cases, cloud providers have special features or unusual requirements. These are outlined in [More Details for Specific Clouds](#).

OS Initial Login and Setup

Once a machine is provisioned, Brooklyn will normally attempt to log in via SSH and configure the machine sensibly.

The credentials for the initial OS log on are typically discovered from the cloud, but in some environments this is not possible. The keys `loginUser` and either `loginUser.password` or `loginUser.privateKeyFile` can be used to force Brooklyn to use specific credentials for the initial login to a cloud-provisioned machine.

(This custom login is particularly useful when using a custom image templates where the cloud-side account management logic is not enabled. For example, a vCloud (vCD) template can have guest customization that will change the root password. This setting tells Apache Brooklyn to only use the given password, rather than the initial randomly generated password that vCD returns. Without this property, there is a race for such templates: does Brooklyn manage to

create the admin user before the guest customization changes the login and reboots, or is the password reset first (the latter means Brooklyn can never ssh to the VM). With this property, Brooklyn will always wait for guest customization to complete before it is able to ssh at all. In such cases, it is also recommended to use `useJcloudsSshInit=false`.)

Following a successful logon, Brooklyn performs the following steps to configure the machine:

1. creates a new user with the same name as the user `brooklyn` is running as locally (this can be overridden with `user`, below).
2. install the local user's `~/.ssh/id_rsa.pub` as an `authorized_keys` on the new machine, to make it easy for the operator to `ssh` in (override with `privateKeyFile`; or if there is no `id_{r,d}sa{,.pub}` an ad hoc keypair will be generated for the regular Brooklyn user; if there is a passphrase on the key, this must be supplied)
3. give `sudo` access to the newly created user (override with `grantUserSudo: false`)
4. disable direct `root` login to the machine

These steps can be skipped or customized as described below.

jclouds Config Keys

The following is a subset of the most commonly used configuration keys used to customize cloud provisioning. For more keys and more detail on the keys below, see [JcloudsLocationConfig](#).

VM Creation

- Most providers require exactly one of either `region` (e.g. `us-east-1`) or `endpoint` (the URL, usually for private cloud deployments)
- Hardware requirements can be specified, including `minRam`, `minCores`, `minDisk` and `os64Bit`; or as a specific `hardwareId`
- VM image constraints can be set using `osFamily` (e.g. `Ubuntu`, `CentOS`, `Debian`, `RHEL`) and `osVersionRegex`, or specific VM images can be specified using `imageId` or `imageNameRegex`
- Specific VM images can be specified using `imageId` or `imageNameRegex`
- Specific Security Groups can be specified using `securityGroups`, as a list of strings (the existing security group names), or `inboundPorts` can be set, as a list of numeric ports (selected clouds only)
- Where a key pair is registered with a target cloud for logging in to machines, Brooklyn can be configured to request this when provisioning VMs by setting `keyPair` (selected clouds only). Note that if this `keyPair` does not correspond your default `~/.ssh/id_rsa`, you must typically also specify the corresponding `loginUser.privateKeyFile` as a file or URL accessible from Brooklyn.
- A specific VM name (often the hostname) base to be used can be specified by setting `groupId`. By default, this name is constructed based on the `entity` which is creating it, including the ID of the app and of the `entity`. (As many cloud portals let you filter views, this can help find a specific `entity` or all machines for a given application.) For more sophisticated control over host naming, you can supply a custom `CloudMachineNamer`, for example `cloudMachineNamer: CustomMachineNamer`. `CustomMachineNamer` will use the `entity`'s name or following a template you supply. On many clouds, a random suffix will be appended to help guarantee uniqueness; this can be removed by setting `vmNameSaltLength: 0` (selected clouds only).
- A DNS domain name where this host should be placed can be specified with `domainName` (in selected clouds only)
- User metadata can be attached using the syntax `userMetadata: { key: value, key2: "value 2" }` (or `userMetadata=key=value,key2="value 2"` in a properties file)

- By default, several pieces of user metadata are set to correlate VMs with Brooklyn entities, prefixed with `brooklyn-`. This user metadata can be omitted by setting `includeBrooklynUserMetadata: false`.
- You can specify the number of attempts Brooklyn should make to create machines with `machineCreateAttempts` (jclouds only). This is useful as an efficient low-level fix for those occasions when cloud providers give machines that are dead on arrival. You can of course also resolve it at a higher level with a [policy](#) such as [ServiceRestarter](#).
- If you want to investigate failures, set `destroyOnFailure: false` to keep failed VM's around. (You'll have to manually clean them up.) The default is false: if a VM fails to start, or is never ssh'able, then the VM will be terminated.
- You can set `useMachinePublicAddressAsPrivateAddress` to true to overwrite the VMs private IP with its public IP. This is useful as it can be difficult to get VMs communicating via the private IPs they are assigned in some clouds. Using this config, blueprints which use private IPs can still be deployed to these clouds.

OS Setup

- `user` and `password` can be used to configure the operating user created on cloud-provisioned machines
- The `loginUser` config key (and subkeys) control the initial user to log in as, in cases where this cannot be discovered from the cloud provider
- Private keys can be specified using `privateKeyFile`; these are not copied to provisioned machines, but are required if using a local public key or a pre-defined `authorized_keys` on the server. (For more information on SSH keys, see [here](#).)
- If there is a passphrase on the key file being used, you must supply it to Brooklyn for it to work, of course! `privateKeyPassphrase` does the trick (as in `brooklyn.location.jclouds.privateKeyPassphrase`, or other places where `privateKeyFile` is valid). If you don't like keys, you can just use a plain old `password`.
- Public keys can be specified using `publicKeyFile`, although these can usually be omitted if they follow the common pattern of being the private key file with the suffix `.pub` appended. (It is useful in the case of `loginUser.publicKeyFile`, where you shouldn't need, or might not even have, the private key of the `root` user when you log in.)
- Provide a list of URLs to public keys in `extraSshPublicKeyUrls`, or the data of one key in `extraSshPublicKeyData`, to have additional public keys added to the `authorized_keys` file for logging in. (This is supported in most but not all locations.)
- Use `dontCreateUser` to have Brooklyn run as the initial `loginUser` (usually `root`), without creating any other user.
- A post-provisioning `setup.script` can be specified to run an additional script, before making the `Location` available to entities. This may take the form of a URL of a script or a [data URI](#). Note that if using a data URI it is usually a good idea to [base64](#) this string to escape problem characters in more complex scripts. The base64 encoded script should then be prefixed with `data:text/plain;base64,` to denote this. For example if you wanted to disable a yum repository called `reponame` prior to using the machine, you could use the following command:

```
sudo yum-config-manager --disable reponame
```

Base64 encoding can be done with a tool such as [this](#) or a linux command such as:

```
echo "sudo yum-config-manager --disable reponame" | base64
```

With the base64 prefix this would then look like this:

```
setup.script: data:text/plain;base64,c3VkbYB5dW0tY29uZmInLW1hbmFnZXIgL1kaXNhYmx1IHJlcG9uYW11
```

The `setup.script` can also take [FreeMarker](#) variables in a `setup.script.vars` property. Variables are set in the format `key1:value1,key2:value2` and used in the form `${key1}`. So for the above example:

```
setup.script.vars: repository:reponame
```

then

```
setup.script: data:sudo yum-config-manager --disable ${repository}
```

or encoded in base64:

```
setup.script: data:text/plain;base64,c3VkybYB5dw0tY29uZmInLW1hbmFnZXIgLlS1kaXNhYmXlICR7cmVwb3NpdG9yeX0=
```

This enables the name of the repository to be passed in to the script.

- Use `openIptables: true` to automatically configure `iptables`, to open the TCP ports required by the software process. One can alternatively use `stopIptables: true` to entirely stop the iptables service.
- Use `Entity` configuration flag `effector.add.openInboundPorts: true` to add an `effector` for opening ports in a cloud Security Group. The config is supported for all `SoftwareProcessImpl` implementations.
- Use `installDevUrandom: true` to fall back to using `/dev/urandom` rather than `/dev/random`. This setting is useful for cloud VMs where there is not enough random entropy, which can cause `/dev/random` to be extremely slow (causing `ssh` to be extremely slow to respond).
- Use `useJcloudsSshInit: false` to disable the use of the native jclouds support for initial commands executed on the VM (e.g. for creating new users, setting root passwords, etc.). Instead, Brooklyn's `ssh` support will be used. Timeouts and retries are more configurable within Brooklyn itself. Therefore this option is particularly recommended when the VM startup is unusual (for example, if guest customizations will cause reboots and/or will change login credentials).
- Use `brooklyn.ssh.config.noDeleteAfterExec: true` to keep scripts on the server after execution. The contents of the scripts and the stdout/stderr of their execution are available in the Brooklyn web console, but sometimes it can also be useful to have them on the box. This setting prevents scripts executed on the VMs from being deleted on completion. Note that some scripts run periodically so this can eventually fill a disk; it should only be used for dev/test.

Custom Template Options

jclouds supports many additional options for configuring how a virtual machine is created and deployed, many of which are for cloud-specific features and enhancements. Brooklyn supports some of these, but if what you are looking for is not supported directly by Brooklyn, we instead offer a mechanism to set any parameter that is supported by the jclouds template options for your cloud.

Part of the process for creating a virtual machine is the creation of a jclouds `TemplateOptions` object. jclouds providers extends this with extra options for each cloud - so when using the AWS provider, the object will be of type `AWSec2TemplateOptions`. By [examining the source code](#), you can see all of the options available to you.

The `templateOptions` config key takes a map. The keys to the map are method names, and Brooklyn will find the method on the `TemplateOptions` instance; it then invokes the method with arguments taken from the map value. If a method takes a single parameter, then simply give the argument as the value of the key; if the method takes multiple parameters, the value of the key should be an array, containing the argument for each parameter.

For example, here is a complete [blueprint](#) that sets some AWS EC2 specific options:

```
location: AWS_eu-west-1
services:
- type: org.apache.brooklyn.entity.software.base.EmptySoftwareProcess
  provisioning.properties:
    templateOptions:
      subnetId: subnet-041c8373
      mapNewVolumeToDeviceName: ["/dev/sda1", 100, true]
      securityGroupIds: ['sg-4db68928']
```

Here you can see that we set three template options:

- `subnetId` is an example of a single parameter method. Brooklyn will effectively try to run the statement `templateOptions.subnetId("subnet-041c88373");`
- `mapNewVolumeToDeviceName` is an example of a multiple parameter method, so the value of the key is an array. Brooklyn will effectively try to run the statement `templateOptions.mapNewVolumeToDeviceName("/dev/sda1", 100, true);`
- `securityGroupIds` demonstrates an ambiguity between the two types; Brooklyn will first try to parse the value as a multiple parameter method, but there is no method that matches this parameter. In this case, Brooklyn will next try to parse the value as a single parameter method which takes a parameter of type `List`; such a method does exist so the operation will succeed.

If the method call cannot be matched to the template options available - for example if you are trying to set an AWS EC2 specific option but your `location` is an OpenStack cloud - then a warning is logged and the option is ignored.

Cloud Machine Naming

The name that Apache Brooklyn generates for your virtual machine will, by default, be based on your Apache Brooklyn server name and the IDs of the entities involved. This is the name you see in places such as the AWS console and will look something like:

```
brooklyn-o8jq14-machinename-rkix-tomcat-wi-nca6-14b
```

If you have created a lot of virtual machines, this kind of naming may not be helpful. This can be changed using the following **YAML** in your `location`'s `brooklyn.config`:

```
cloudMachineNamer: org.apache.brooklyn.core.location.cloud.names.CustomMachineNamer
custom.machine.namer.machine: My-Custom-Name-${entity.displayName}
```

A **FreeMarker** format is used in `custom.machine.namer.machine` which can take values from places such as the launching `entity` or `location`.

The above example will create a name such as:

```
My-Custom-Name-Tomcat
```

Allowing you to more easily identify your virtual machines.

More Details on Specific Clouds

Clouds vary in the format of the identity, credential, endpoint, and region. Some also have their own idiosyncracies. More details for configuring some common clouds is included below. You may also find these sources helpful:

- The **template brooklyn.properties** file in the Getting Started guide contains numerous examples of configuring specific clouds, including the format of credentials and options for sometimes-fiddly private clouds.
- The **jclouds guides** describes low-level configuration sometimes required for various clouds.

Amazon Web Services (AWS)

Credentials

AWS has an "access key" and a "secret key", which correspond to Brooklyn's identity and credential respectively.

These keys are the way for any programmatic mechanism to access the AWS API.

To generate an access key and a secret key, see [jclouds instructions](#) and [AWS IAM instructions](#).

An example of the expected format is shown below:

```
location:
  jclouds:aws-ec2:
    region: us-east-1
    identity: ABCDEFGHIJKLMNOPQRST
    credential: abcdefghijklmnopqrstu+vwxyzabcdefghijklmnop
```

Users are strongly recommended to use [externalized configuration](#) for better credential management, for example using [Vault](#).

Common Configuration Options

Below are examples of configuration options that use values specific to AWS EC2:

- The `region` is the [AWS region code](#). For example, `region: us-east-1`. You can in-line the region name using the following format: `jclouds:aws-ec2:us-east-1`. A specific availability zone within the region can be specified by including its letter identifier as a suffix. For example, `region: us-east-1a`.
- The `hardwareId` is the [instance type](#). For example, `hardwareId: m4.large`.
- The `imageId` is the region-specific [AMI id](#). For example, `imageId: us-east-1/ami-05ebd06c`.
- The `securityGroups` option takes one or more names of pre-existing [security groups](#). For example, `securityGroups: mygroup1` Or `securityGroups: [mygroup1, mygroup2]`.

Using Subnets and Security Groups

Apache Brooklyn can run with AWS VPC and both public and private subnets. Simply provide the `subnet-a1b2c3d4` as the `networkName` when deploying:

```
location:
  jclouds:aws-ec2:
    region: us-west-1
    networkName: subnet-a1b2c3d4 # use your subnet ID
```

Subnets are typically used in conjunction with security groups. Brooklyn does *not* attempt to open additional ports when private subnets or security groups are supplied, so the subnet and ports must be configured appropriately for the blueprints being deployed. You can configure a default security group with appropriate (or all) ports opened for access from the appropriate (or all) CIDRs and security groups, or you can define specific `securityGroups` on the `location` or as `provisioning.properties` on the entities.

Make sure that Brooklyn has access to the machines under management. This includes SSH, which might be done with a public IP created with inbound access on port 22 permitted for a CIDR range including the IP from which Brooklyn contacts it. Alternatively you can run Brooklyn on a machine in that same subnet, or set up a VPN or jump host which Brooklyn will use.

EC2 "Classic" Problems with VPC-only Hardware Instance Types

If you have a pre-2014 Amazon account, it is likely configured in some regions to run in "EC2 Classic" mode by default, instead of the more modern "VPC" default mode. This can cause failures when requesting certain hardware configurations because many of the more recent hardware "instance types" only run in "VPC" mode. For instance when requesting an instance with `minRam: 8gb`, Brooklyn may opt for an `m4.large`, which is a VPC-only instance type. If you are in a region configured to use "EC2 Classic" mode, you may see a message such as this:

```
400 VPCResourceNotSpecified: The specified instance type can only be used in a VPC.
A subnet ID or network interface ID is required to carry out the request.
```

This is a limitation of "legacy" accounts. The easiest fixes are either:

- specify an instance type which is supported in classic, such as `m3.xlarge` (see below)
- move to a different region where VPC is the default (`eu-central-1` should work as it *only* offers VPC mode, irrespective of the age of your AWS account)
- get a new AWS account -- "VPC" will be the default mode (Amazon recommend this and if you want to migrate existing deployments they provide [detailed instructions](#))

To understand the situation, the following resources may be useful:

- Background on VPC vs Classic: <http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-vpc.html>
- Good succinct answers to FAQs: http://aws.amazon.com/vpc/faqs/#Default_VPCs
- Check if a region in your account is "VPC" or "Classic":
<http://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/default-vpc.html#default-vpc-availability>
- Regarding instance types:
 - All instance types: <https://aws.amazon.com/ec2/instance-types>
 - Those which require VPC: <http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-vpc.html#vpc-only-instance-types>

If you want to solve this problem with your existing account, you can create a VPC and instruct Brooklyn to use it:

1. Use the "Start VPC Wizard" option in [the VPC dashboard](#), making sure it is for the right region, and selecting a "Single Public Subnet". (More information is in [these AWS instructions](#).)
2. Once the VPC is created, open the "Subnets" view and modify the "Public subnet" so that it will "Auto-assign Public IP".
3. Next click on the "Security Groups" and find the `default` security group for that VPC. Modify its "Inbound Rules" to allow "All traffic" from "Anywhere". (Or for more secure options, see the instructions in the previous section, "Using Subnets".)
4. Finally make a note of the subnet ID (e.g. `subnet-a1b2c3d4`) for use in Brooklyn.

You can then deploy blueprints to the subnet, allowing VPC hardware instance types, by specifying the subnet ID as the `networkName` in your [YAML blueprint](#). This is covered in the previous section, "Using Subnets".

Tidying up after jclouds

Security groups are not always deleted by jclouds. This is due to a limitation in AWS (see <https://issues.apache.org/jira/browse/JCLOUDS-207>). In brief, AWS prevents the security group from being deleted until there are no VMs using it. However, there is eventual consistency for recording which VMs still reference those security groups: after deleting the VM, it can sometimes take several minutes before the security group can be deleted. jclouds retries for 3 seconds, but does not block for longer.

Whilst there is eventual consistency for recording which VMs still reference security groups, after deleting a VM, it can sometimes take several minutes before a security group can be deleted

There is utility written by [Cloudsoft](#) for deleting these unused resources:

<http://blog.abstractvisitorpattern.co.uk/2013/03/tidying-up-after-jclouds.html>.

Azure Compute ARM

Azure Resource Manager (ARM) is a framework for deploying and managing applications across resources and managing groups of resources as single logical units on the Microsoft Azure cloud computing platform.

Setup the Azure credentials

Azure CLI 2.0

Firstly, install and configure Azure CLI following [these steps](#).

You will need to obtain your *subscription ID* and *tenant ID* from Azure. To do this using the CLI, first, log in:

```
az login
```

Or, if you are already logged in, request an account listing:

```
az account list
```

In either case, this will return a subscription listing, similar to that shown below.

```
[
  {
    "cloudName": "AzureCloud",
    "id": "012e832d-XXXX-XXXX-XXXX-XXXXXXXXXXXX",
    "isDefault": true,
    "name": "QA Team",
    "state": "Enabled",
    "tenantId": "ba85e8cd-XXXX-XXXX-XXXX-XXXXXXXXXXXX",
    "user": {
      "name": "qa@example.com",
      "type": "user"
    }
  },
  {
    "cloudName": "AzureCloud",
    "id": "341751b0-XXXX-XXXX-XXXX-XXXXXXXXXXXX",
    "isDefault": false,
    "name": "Developer Team",
    "state": "Enabled",
    "tenantId": "ba85e8cd-XXXX-XXXX-XXXX-XXXXXXXXXXXX",
    "user": {
      "name": "dev@example.com",
      "type": "user"
    }
  }
]
```

Choose one of the subscriptions and make a note of its *id* - henceforth the subscription ID - and the *tenantId*.

Next we need to create an *application* and a *service principle*, and grant permissions to the service principle. Use these commands:

```
# Create an AAD application with your information.
az ad app create --display-name <name> --password <Password> --homepage <home-page> --identifier-uri <identifier-uri>

# For example: az ad app create --display-name "myappname" --password abcd --homepage "https://myappwebsite" --identifier-uri "https://myappwebsite"
```

Take a note of the *appId* shown.

```
# Create a Service Principal
az ad sp create --id <Application-id>
```

Take a note of the *objectId* shown - this will be the service principal object ID. (Note that any of the *servicePrincipalNames* can also be used in place of the object ID.)

```
# Assign roles for this service principal. The "principal" can be the "objectId" or any one of the "servicePrincipalNames" from the previous step
az role assignment create --assignee <Service-Principal> --role Contributor --scope /subscriptions/<Subscription-ID>/
```

By this stage you should have the following information:

- A subscription ID
- A tenant ID
- An application ID
- A service principle (either by its object ID, or by any one of its names)

We can now verify this information that this information can be used to log in to Azure:

```
az login --service-principal -u <Application-ID> --password abcd --tenant <Tenant-ID>
```

Azure CLI 1.0

Firstly, install and configure Azure CLI following [these steps](#).

Using the Azure CLI, run the following commands to create a service principal

```
# Set mode to ARM
azure config mode arm

# Enter your Microsoft account credentials when prompted
azure login

# Set current subscription to create a service principal
azure account set <Subscription-id>

# Create an AAD application with your information.
azure ad app create --name <name> --password <Password> --home-page <home-page> --identifier-uri <identifier-uri>

# For example: azure ad app create --name "myappname" --password abcd --home-page "https://myappwebsite" --identifier-uri "https://myappwebsite"

# Output will include a value for `Application Id`, which will be used for the live tests

# Create a Service Principal
azure ad sp create --applicationId <Application-id>

# Output will include a value for `Object Id`, to be used in the next step
```

Run the following commands to assign roles to the service principal

```
# Assign roles for this service principal
azure role assignment create --objectId <Object-id> -o Contributor -c /subscriptions/<Subscription-id>/
```

Look up the the tenant Id

```
azure account show -s <Subscription-id> --json

# output will be a JSON which will include the `Tenant id`
```

Verify service principal

```
azure login -u <Application-id> -p <Password> --service-principal --tenant <Tenant-id>
```

Using the Azure ARM Location

Below is an example Azure ARM [location](#) in [YAML](#) which will launch a Ubuntu instance in south east asia:

```
brooklyn.catalog:
  id: my-azure-arm-location
  name: "My Azure ARM location"
  itemType: location
  item:
    type: jclouds:azurecompute-arm
    brooklyn.config:
      identity: <Application-id>
      credential: <Password>
      endpoint: https://management.azure.com/subscriptions/<Subscription-id>
      oauth.endpoint: https://login.microsoftonline.com/<Tenant-id>/oauth2/token

    jclouds.azurecompute.arm.publishers: OpenLogic
    region: southeastasia
    loginUser: brooklyn
    templateOptions:
      overrideAuthenticateSudo: true
```

Fill the values `<Application-id>`, `<Password>`, `<Subscription-id>` and `<Tenant-id>` in from the values generated when setting up your credentials. In addition; several keys, not required in other locations need to be specified in order to use the Azure Compute ARM [location](#). These are:

```
jclouds.azurecompute.arm.publishers: OpenLogic
```

The publishers is any item from the list available here: <https://docs.microsoft.com/en-us/azure/virtual-machines/virtual-machines-linux-cli-ps-findimage>

```
region: southeastasia
```

The region is any region from the list available here: <https://azure.microsoft.com/en-us/regions/>

```
loginUser: brooklyn
```

The loginUser can be anything, as long as it's specified.

```
templateOptions:
  overrideAuthenticateSudo: true
```

The `overrideAuthenticateSudo: true` key tells Apache Brooklyn that default on Azure images do not have passwordless sudo configured by default.

Useful configuration options for Azure ARM

You can add these options directly under the `brooklyn.config` element in the example above:

- `jclouds.compute.resourcename-prefix` and `jclouds.compute.resourcename-delimiter` - defaults to `jclouds` and `-` respectively. If jclouds is choosing the name for a resource (for example, a virtual machine), these properties will alter the way the resource is named.

You can add these options into the `templateOptions` element inside the `brooklyn.config` element in the example above:

- `resourceGroup` - select a Resource Group to deploy resources into. If not given, jclouds will generate a new resource group with a partly-random name.

Using Windows on Azure ARM

This section contains material how to create a Windows [location](#) on Azure ARM. Some of the used parameters are explained in the section above.

Windows on Azure ARM requires manually created [Azure KeyVault](#). Azure KeyVaults can be created [via Azure cli](#) or [Azure portal UI](#). KeyVault's secret is a key stored in protected .PFX file. It needs to be prepared upfront or created with the `Add-AzureKeyVaultKey` cmdlet.

- `osFamily: windows` tells Apache Brooklyn to consider it as a Windows machine
- `useJcloudsSshInit: false` tells jclouds to not try to connect to the VM
- `vmNameMaxLength: 15` tells the cloud client to strip the VM name to maximum 15 characters. This is the maximum size supported by Azure Windows VMs.
- `winrm.useHttps` tells Apache Brooklyn to configure the WinRM client to use HTTPS.
- `secrets` Specifies the KeyVault configuration
 - `sourceVault` Resource `id` of the KeyVault
 - `vaultCertificates` `certificateStore` has to use `My` as a value. KeyVault's `certificateUrl` . An URI to the [Secret Identifier](#)
- `windowsConfiguration`
 - `provisionVMagent` whether Azure to install an agent on the VM. It must be set to `true`
 - `winRM` It defines the `listeners` section. If `listeners` is `https` then `certificateUrl` needs to be set. Its value must match the one of `secrets`'s `certificateUrl` .
- `additionalUnattendContent` Additional content. Normally it can be defined as `null`
- `enableAutomaticUpdates` whether to enable the automatic windows updates. It can be set to `false` , if automatic updates are not desired

Sample Windows Blueprint

Placeholders surrounded with `<>` have to be replaced with their respective values.

```
brooklyn.catalog:
  id: my-azure-arm-location
  name: "My Azure ARM location"
  itemType: location
  item:
    type: jclouds:azurecompute-arm
    brooklyn.config:
      identity: <Application-id>
      credential: <Password>
      endpoint: https://management.azure.com/subscriptions/<Subscription-id>
      oauth.endpoint: https://login.microsoftonline.com/<Tenant-id>/oauth2/token
      jclouds.azurecompute.arm.publishers: MicrosoftWindowsServer
      jclouds.azurecompute.operation.timeout: 120000

      winrm.useHttps: true
      osFamily: windows
      imageId: <Azure_location>/MicrosoftWindowsServer/WindowsServer/2012-R2-Datacenter
      region: <Azure_location>
      vmNameMaxLength: 15
      useJcloudsSshInit: false
      destroyOnFailure: false

    templateOptions:
      overrideLoginUser: brooklyn
      overrideLoginPassword: "secretPass1!"
      secrets:
        - sourceVault:
            id: "/subscriptions/<Subscription-id>/resourceGroups/<ResourceGroup>/providers/Microsoft.KeyVault/vaults/<KeyVault-name>"
            vaultCertificates:
              - certificateUrl: "<KeyVault-uri>"
                certificateStore: My
            windowsConfiguration:
              provisionVMAgent: true
            winRM:
              listeners:
                - protocol: https
                  certificateUrl: "<KeyVault-uri>"
              additionalUnattendContent: null
              enableAutomaticUpdates: true
```

Known issues

There are currently two known issues with Azure ARM:

- It can take a long time for VMs to be provisioned
- The Azure ARM APIs appear to have some fairly strict rate limiting that can result in `AzureComputeRateLimitExceededException`

Azure Compute Classic

Azure is a cloud computing platform and infrastructure created by Microsoft. Apache Brooklyn includes support for both Azure Classic and Azure ARM, as one of the [Apache jclouds](#) supported clouds `Microsoft Azure Compute`.

The two modes of using Azure are the "classic deployment" model and the newer "Azure Resource Manager" (ARM) model. See <https://azure.microsoft.com/en-gb/documentation/articles/resource-manager-deployment-model/> for details.

Setup the Azure credentials

Microsoft Azure requests are signed by SSL certificate. You need to upload one into your account in order to use an Azure [location](#).

```
# create the certificate request
mkdir -m 700 $HOME/.brooklyn
openssl req -x509 -nodes -days 365 -newkey rsa:1024 -keyout $HOME/.brooklyn/azure.pem -out $HOME/.brooklyn/azure.pem
# create the p12 file, and note your export password. This will be your test credentials.
openssl pkcs12 -export -out $HOME/.brooklyn/azure.p12 -in $HOME/.brooklyn/azure.pem -name "brooklyn :: $USER"
# create a cer file
openssl x509 -inform pem -in $HOME/.brooklyn/azure.pem -outform der -out $HOME/.brooklyn/azure.cer
```

Finally, upload .cer file to the management console at

<https://manage.windowsazure.com/@myId#Workspaces/AdminTasks/ListManagementCertificates> to authorize this certificate.

Please note, you can find the "myId" value for this link by looking at the URL when logged into the Azure management portal.

Note, you will need to use `.p12` format in the `brooklyn.properties`.

How to configure Apache Brooklyn to use Azure Compute

First, in your `brooklyn.properties` define a [location](#) as follows:

```
brooklyn.location.jclouds.azurecompute.identity=$HOME/.brooklyn/azure.p12
brooklyn.location.jclouds.azurecompute.credential=<P12_EXPORT_PASSWORD>
brooklyn.location.jclouds.azurecompute.endpoint=https://management.core.windows.net/<YOUR_SUBSCRIPTION_ID>
brooklyn.location.jclouds.azurecompute.vmNameMaxLength=45
brooklyn.location.jclouds.azurecompute.jclouds.azurecompute.operation.timeout=120000
brooklyn.location.jclouds.azurecompute.user=<USER_NAME>
brooklyn.location.jclouds.azurecompute.password=<PASSWORD>
```

During the VM provisioning, Azure will set up the account with `<USER_NAME>` and `<PASSWORD>` automatically. Notice, `<PASSWORD>` must be a minimum of 8 characters and must contain 3 of the following: a lowercase character, an uppercase character, a number, a special character.

To force Apache Brooklyn to use a particular image in Azure, say Ubuntu 14.04.1 64bit, one can add:

```
brooklyn.location.jclouds.azurecompute.imageId=b39f27a8b8c64d52b05eac6a62ebad85__Ubuntu-14_04_1-LTS-amd64-server-20150123-en-us-30GB
```

From `$BROOKLYN_HOME`, you can list the image IDs available using the following command:

```
./bin/client "list-images --location azure-west-europe"
```

To force Brooklyn to use a particular hardwareSpec in Azure, one can add something like:

```
brooklyn.location.jclouds.azurecompute.hardwareId=BASIC_A2
```

From `$BROOKLYN_HOME`, you can list the hardware profile IDs available using the following command:

```
./bin/client "list-hardware-profiles --location azure-west-europe"
```

At the time of writing, the classic deployment model has the possible values shown below. See <https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-size-specs/> for further details, though that description focuses on the new "resource manager deployment" rather than "classic".

- `Basic_A0` to `Basic_A4`
- `Standard_D1` to `Standard_D4`
- `Standard_G1` to `Standard_G5`
- `ExtraSmall` , `Small` , `Medium` , `Large` , `ExtraLarge`

Named location

For convenience, you can define a named `location`, like:

```
brooklyn.location.named.azure-west-europe=jclouds:azurecompute:West Europe
brooklyn.location.named.azure-west-europe.displayName=Azure West Europe
brooklyn.location.named.azure-west-europe.imageId=b39f27a8b8c64d52b05eac6a62ebad85__Ubuntu-14_04_1-LTS-amd64-se
rver-20150123-en-us-30GB
brooklyn.location.named.azure-west-europe.hardwareId=BASIC_A2
brooklyn.location.named.azure-west-europe.user=test
brooklyn.location.named.azure-west-europe.password=MyPassword1!
```

This will create a `location` named `azure-west-europe` . It will inherit all the configuration defined on `brooklyn.location.jclouds.azurecompute` . It will also augment and override this configuration (e.g. setting the display name, image id and hardware id).

On Linux VMs, The `user` and `password` will create a user with that name and set its password, disabling the normal login user and password defined on the `azurecompute` `location`.

Windows VMs on Azure

The following configuration options are important for provisioning Windows VMs in Azure:

- `osFamily: windows` tells Apache Brooklyn to consider it as a Windows machine
- `useJcloudsSshInit: false` tells jclouds to not try to connect to the VM
- `vmNameMaxLength: 15` tells the cloud client to strip the VM name to maximum 15 characters. This is the maximum size supported by Azure Windows VMs.
- `winrm.useHttps` tells Apache Brooklyn to configure the WinRM client to use HTTPS.

This is currently not supported in the default configuration for other clouds, where Apache Brooklyn is deploying Windows VMs.

If the parameter value is `false` the default WinRM port is 5985; if `true` the default port for WinRM will be 5986. Use of default ports is strongly recommended.

- `winrm.useNtlm` tells Apache Brooklyn to configure the WinRM client to use NTLM protocol.

For Azure, this is mandatory.

For other clouds, this value is used in the cloud init script to configure WinRM on the VM.

If the value is `true` then Basic Authentication will be disabled and the WinRM client will only use Negotiate plus NTLM.

If the value is `false` then Basic Authentication will be enabled and the WinRM client will use Basic Authentication.

NTLM is the default Authentication Protocol.

The format of this configuration option is subject to change: WinRM supports several authentication mechanisms, so this may be changed to a prioritised list so as to provide fallback options.

- `user` tells Apache Brooklyn which user to login as. The value should match that supplied in the `overrideLoginUser` of the `templateOptions`.
- `password` : tells Apache Brooklyn the password to use when connecting. The value should match that supplied in the `overrideLoginPassword` of the `templateOptions`.
- `templateOptions: { overrideLoginUser: adminuser, overrideLoginPassword: Pa55w0rd! }` tells the Azure Cloud to provision a VM with the given admin username and password. Note that no "Administrator" user will be created.

If this config is not set then the VM will have a default user named "jclouds" with password "Azur3Compute!". It is **Strongly Recommended** that these template options are set.

Notice: one cannot use `Administrator` as the user in Azure.

This configuration is subject to change in future releases.

Sample Windows Blueprint

Below is an example for provisioning a Windows-based `entity` on Azure. Note the placeholder values for the identity, credential and password.

```
name: Windows Test @ Azure
location:
  jclouds:azurecompute:West Europe:
    identity: /home/users/brooklyn/.brooklyn/azure.p12
    credential: xxxxxxxp12
    endpoint: https://management.core.windows.net/12345678-1234-1234-1234-123456789abc
    imageId: 3a50f22b388a4ff7ab41029918570fa6__Windows-Server-2012-Essentials-20141204-enus
    hardwareId: BASIC_A2
    osFamily: windows
    useJcloudsSshInit: false
    vmNameMaxLength: 15
    winrm.useHttps: true
    user: brooklyn
    password: secretPass1!
    templateOptions:
      overrideLoginUser: brooklyn
      overrideLoginPassword: secretPass1!
services:
- type: org.apache.brooklyn.entity.software.base.VanillaWindowsProcess
  brooklyn.config:
    install.command: echo install phase
    launch.command: echo launch phase
    checkRunning.command: echo launch phase
```

Below is an example named `location` for Azure, configured in `brooklyn.properties`. Note the placeholder values for the identity, credential and password.

```
brooklyn.location.named.myazure=jclouds:azurecompute:West Europe
brooklyn.location.named.myazure.displayName=Azure West Europe (windows)
brooklyn.location.named.myazure.identity=$HOME/.brooklyn/azure.p12
brooklyn.location.named.myazure.credential=<P12_EXPORT_PASSWORD>
brooklyn.location.named.myazure.endpoint=https://management.core.windows.net/<YOUR_SUBSCRIPTION_ID>
brooklyn.location.named.myazure.vmNameMaxLength=15
brooklyn.location.named.myazure.jclouds.azurecompute.operation.timeout=120000
brooklyn.location.named.myazure.imageId=3a50f22b388a4ff7ab41029918570fa6__Windows-Server-2012-Essentials-20141204-enus
brooklyn.location.named.myazure.hardwareId=BASIC_A2
brooklyn.location.named.myazure.osFamily=windows
```



```
brooklyn.location.named.myazure.useJcloudsSshInit=false
brooklyn.location.named.myazure.winrm.useHttps=true
brooklyn.location.named.myazure.user=brooklyn
brooklyn.location.named.myazure.password=secretPass1!
brooklyn.location.named.myazure.templateOptions={ overrideLoginUser: amp, overrideLoginPassword: secretPass1! }
```

User and Password Configuration

As described under the configuration options, the username and password must be explicitly supplied in the configuration.

This is passed to the Azure Cloud during provisioning, to create the required user. These values correspond to the options `AdminUsername` and `AdminPassword` in the Azure API.

If a hard-coded password is not desired, then within Java code a random password could be auto-generated and passed into the call to `location.obtain(Map<?,?>)` to override these values.

This approach differs from the behaviour of clouds like AWS, where the password is auto-generated by the cloud provider and is then retrieved via the cloud provider's API after provisioning the VM.

WinRM Configuration

The WinRM initialization in Azure is achieved through configuration options in the VM provisioning request. The required configuration is to enable HTTPS (if Azure is told to use http, the VM comes pre-configured with WinRM encrypted over HTTP). The default is then to support NTLM protocol.

The setup of Windows VMs on Azure differs from that on other clouds, such as AWS. In contrast, on AWS an init script is passed to the cloud API to configure WinRM appropriately.

Windows initialization scripts in Azure are unfortunately not supported in "classic deployment" model, but are available in the newer "resource manager deployment" model as an "Azure VM Extension".

Apache CloudStack

Connection Details

The endpoint URI will normally have the suffix `/client/api/`.

The identity is the "api key" and the credential is the "secret key". These can be generated in the CloudStack gui: under accounts, select "view users", then "generate key".

```
location:
  jclouds:cloudstack:
    endpoint: https://cloud.acme.com/client/api
    identity: abcdefghijklmnopqrstuvwxyz01234567890-abcdefghijklmnopqrstuvwxyz01234567890-abcdefghij
    credential: mycred-abcdefghijklmnopqrstuvwxyz01234567890-abcdefghijklmnopqrstuvwxyz01234567890-abc
```

Users are strongly recommended to use [externalized configuration](#) for better credential management, for example using [Vault](#).

Common Configuration Options

Below are examples of configuration options that use values specific to CloudStack environments:

- The `imageId` is the template id. For example, `imageId: db0bcce3-9e9e-4a87-a953-2f46b603498f`.

- The `region` is CloudStack zone id. For example `region: 84539b9c-078e-458a-ae26-c3ffc5bb1ec9 ..`
- `networkName` is the network id (within the given zone) to be used. For example, `networkName: 961c03d4-9828-4037-9f4d-3dd597f60c4f .`

For further configuration options, consult [jclouds CloudStack template options](#). These can be used with the `templateOptions` configuration option.

Using a Pre-existing Key Pair

The configuration below uses a pre-existing key pair:

```
location:
  jclouds:cloudstack:
    ...
    loginUser: root
    loginUser.privateKeyFile: /path/to/keypair.pem
    keyPair: my-keypair
```

Using Pre-existing Security Groups

To specify existing security groups, their IDs must be used rather than their names (note this differs from the configuration on other clouds!).

The configuration below uses a pre-existing security group:

```
location:
  jclouds:cloudstack:
    ...
    templateOptions:
      generateSecurityGroup: false
      securityGroupIds:
        - 12345678-90ab-def0-1234-567890abcdef
```

Using Static NAT

Assigning a public IP to a VM at provision-time is referred to as "static NAT" in CloudStack parlance. To give some consistency across different clouds, the configuration option is named `autoAssignFloatingIp` . For example,

```
autoAssignFloatingIp: false .
```

CloudMonkey CLI

The [CloudStack CloudMonkey CLI](#) is a very useful tool. It gives is an easy way to validate that credentials are correct, and to query the API to find the correct zone IDs etc.

Useful commands include:

```
# for finding the ids of the zones:
cloudmonkey api listZones

# for finding the ids of the networks.
cloudmonkey api listNetworks | grep -E "id =|name =|======"
```

CloudStack Troubleshooting

These troubleshooting tips are more geared towards problems encountered in old test/dev CloudStack environment.

Resource Garbage Collection Issues

The environment may run out of resources, due to GC issues, preventing the user from creating new VMs or allocating IP addresses (May respond with this error message: `errorCode=INTERNAL_ERROR, errorText=Job failed due to exception unable to create a deployment for vm`). There are two options worth checking it to enforce clearing up the zombie resources:

- Go to the Accounts tab in the webconsole and tap on the Update Resource Count button.
- Restart the VPC in question from the Network tab.

Releasing Allocated Public IP Addresses

Releasing an allocated Public IP from the web console did not free up the resources. Instead CloudMonkey can be used to dissociate IPs and expunge VMs.

Here is a CloudMonkey script to dissociate any zombie IPs:

```
cloudmonkey set display json;
cloudmonkey api listPublicIpAddresses | grep '"id":' > ips.txt;
sed -i -e s/'      "id": ''/''/g ips.txt;
sed -i -e s/','/''/g ips.txt
for line in $(cat ips.txt); do cloudmonkey api disassociateIpAddress id="$line"; done
rm ips.txt;
cloudmonkey set display default;
```

Restarting VPCs

Errors have been encountered when a zone failed to provision new VMs, with messages like:

```
Job failed due to exception Resource [Host:15] is unreachable: Host 15: Unable to start instance due to null
```

The workaround was to restart the VPC networks:

- Log into the CloudStack web-console.
- Go to Network -> VPC (from the "select view")
- For each of the VPCs, click on the "+" in the "quickview" column, and invoke "restart VPC".

Other symptoms of this issue were that: 1) an administrator could still provision VMs using the admin account, which used a different network; and 2) the host number was very low, so it was likely to be a system host/VM that was faulty.

Google Compute Engine (GCE)

Credentials

GCE uses a service account e-mail address for the identity and a private key as the credential.

To obtain credentials for GCE, use the GCE web page's "APIs & auth -> Credentials" page, creating a "Service Account" of type JSON, then extracting the client_email as the identity and private_key as the credential. For more information, see the [jclouds instructions](#).

An example of the expected format is shown below. Note that when supplying the credential in a properties file, it can either be one long line with `\n` representing the new line characters, or in [YAML](#) it can be split over multiple lines as below:

[illegible]

It is also possible to have the credential be the path of a local file that contains the key. However, this can make it harder to setup and manage multiple Brooklyn servers (particularly when using high availability mode).

Users are strongly recommended to use [externalized configuration](#) for better credential management, for example using [Vault](#).

Quotas

GCE accounts can have low default **quotas**.

It is easy to request a quota increase by submitting a [quota increase form](#).

Networks

GCE accounts often have a limit to the number of networks that can be created. One work around is to manually create a network with the required open ports, and to refer to that named network in Brooklyn's `location` configuration.

To create a network, see [GCE network instructions](#).

For example, for dev/demo purposes an "everything" network could be created that opens all ports.

|| Name || everything | || Description || opens all tcp ports | || Source IP Ranges || 0.0.0.0/0 | || Allowed protocols and ports || tcp:0-65535 and udp:0-65535 |

To configure the `location` to use this, you can include a `location` configuration option like:

```
templateOptions:
  network: https://www.googleapis.com/compute/v1/projects/<project name>/global/networks/everything
```

IBM SoftLayer

Credentials

Credentials can be obtained from the Softlayer API, under "administrative -> user administration -> api-access".

For example:

```
location:
  jclouds:softlayer:
    region: ams01
    identity: my-user-name
    credential: 1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef
```

Users are strongly recommended to use `externalized configuration` for better credential management, for example using `Vault`.

Common Configuration Options

Below are examples of configuration options that use values specific to Softlayer:

- The `region` is the `Softlayer datacenter`. For example, `region: dal05`.
- The `hardwareId` is an auto-generated combination of the hardware configuration options. This is because there is no concept of hardwareId or hardware profile names in Softlayer. An example value is `hardwareId: "cpu=1,memory=1024,disk=25,type=LOCAL"`.
- The `imageId` is the `Image template`. For example, `imageId: CENTOS_6_64`.

VLAN Selection

SoftLayer may provision VMs in different VLANs, even within the same region. Some applications require VMs to be on the *same* internal subnet; blueprints for these can specify this behaviour in SoftLayer in one of two ways.

The VLAN ID can be set explicitly using the fields `primaryNetworkComponentNetworkVlanId` and `primaryBackendNetworkComponentNetworkVlanId` of `SoftLayerTemplateOptions` when specifying the `location` being used in the `blueprint`, as follows:

```
location:
  jclouds:softlayer:
    region: ams01
    templateOptions:
      # Enter your preferred network IDs
      primaryNetworkComponentNetworkVlanId: 1153481
      primaryBackendNetworkComponentNetworkVlanId: 1153483
```

This method requires that a VM already exist and you look up the IDs of its VLANs, for example in the SoftLayer console UI, and that subsequently at least one VM in that VLAN is kept around. If all VMs on a VLAN are destroyed SoftLayer may destroy the VLAN. Creating VLANs directly and then specifying them as IDs here may not work. Add a line note

The second method tells Brooklyn to discover VLAN information automatically: it will provision one VM first, and use the VLAN information from it when provisioning subsequent machines. This ensures that all VMs are on the same subnet without requiring any manual VLAN referencing, making it very easy for end-users.

To use this method, we tell Brooklyn to use `SoftLayerSameVlanLocationCustomizer` as a `location` customizer. This can be done on a `location` as follows:

```
location:
  jclouds:softlayer:
    region: lon02
    customizers:
      - $brooklyn:object:
          type: org.apache.brooklyn.location.jclouds.softlayer.SoftLayerSameVlanLocationCustomizer
          softlayer.vlan.scopeUid: "my-custom-scope"
          softlayer.vlan.timeout: 10m
```

Usually you will want the scope to be unique to a single application, but if you need multiple applications to share the same VLAN, simply configure them with the same scope identifier.

It is also possible with many blueprints to specify this as one of the `provisioning.properties` on an *application*:

```
services:
- type: org.apache.brooklyn.entity.stock.BasicApplication
  id: same-vlan-application
  brooklyn.config:
    provisioning.properties:
      customizers:
        - $brooklyn:object:
            type: org.apache.brooklyn.location.jclouds.softlayer.SoftLayerSameVlanLocationCustomizer
            softlayer.vlan.scopeUid: "my-custom-scope"
            softlayer.vlan.timeout: 10m
```

If you are writing an `entity` in Java, you can also use the helper method `forScope(String)` to create the customizer. Configure the provisioning flags as follows:

```
JcloudsLocationCustomizer vlans = SoftLayerSameVlanLocationCustomizer.forScope("my-custom-scope");
flags.put(JcloudsLocationConfig.JCLOUDS_LOCATION_CUSTOMIZERS.getName(), ImmutableList.of(vlans));
```

Configuration Options

The allowed configuration keys for the `SoftLayerSameVlanLocationCustomizer` are:

- **softlayer.vlan.scopeUid** The scope identifier for locations whose VMs will have the same VLAN.
- **softlayer.vlan.timeout** The amount of time to wait for a VM to be configured before timing out without setting the VLAN ids.
- **softlayer.vlan.publicId** A specific public VLAN ID to use for the specified scope.
- **softlayer.vlan.privateId** A specific private VLAN ID to use for the specified scope.

An `entity` being deployed to a customized `location` will have the VLAN ids set as sensors, with the same names as the last two configuration keys.

NOTE If the SoftLayer `location` is already configured with specific VLANs then this customizer will have no effect.

OpenStack

Apache jclouds

Support for OpenStack is provided by [Apache jclouds](#). For more information, see their guide [here](#).

Connection Details

The endpoint URI is that of keystone (normally on port 5000).

The identity normally consists of a colon-separated tenant and username. The credential is the password. For example:

```
location:
  jclouds:openstack-nova:
    endpoint: http://x.x.x.x:5000/v2.0/
    identity: "your-tenant:your-username"
    credential: your-password
```

OpenStack Nova access information can be downloaded from the openstack web interface, for example as an openrc.sh file. It is usually available from API Access tab in "Access & Security" section. This file will normally contain the identity and credential.

Users are strongly recommended to use [externalized configuration](#) for better credential management, for example using [Vault](#).

Common Configuration Options

Below are examples of configuration options that use values specific to OpenStack environments:

- The `imageId` is the id of an image. For example, `imageId: RegionOne/08086159-8b0b-4970-b332-a7a929ee601f`. These ids can be found from the the CLI or the web-console, for example in IBM Blue Box London, the URL is <https://tenant-region.openstack.blueboxgrid.com/project/images/>.
- The `hardwareId` is the [flavor id](#). For example `hardwareId: RegionOne/1`. These ids can be found from the the CLI or the web-console, for example in IBM Blue Box, the URL is <https://tenant-region.openstack.blueboxgrid.com/admin/flavors/>.

The default flavors are shown below (though the set of flavors can be [managed by the admin](#)):

```
+-----+-----+-----+-----+
| ID | Name      | Memory_MB | Disk |
+-----+-----+-----+-----+
| 1  | m1.tiny   | 512       | 1    |
| 2  | m1.small  | 2048      | 20   |
| 3  | m1.medium | 4096      | 40   |
| 4  | m1.large  | 8192      | 80   |
| 5  | m1.xlarge | 16384     | 160  |
+-----+-----+-----+-----+
```

For further configuration options, consult [jclouds Nova template options](#). These can be used with the [templateOptions](#) configuration option.

Networks

When multiple networks are available you should indicate which ones machines should join. Do this by setting the desired values id as an option in the **templateOptions** configuration:

```
location:
  jclouds:openstack-nova:
    ...
  templateOptions:
    # Assign the node to all networks in the list.
    networks:
      - network-one-id
      - network-two-id
      - ...
```

Floating IPs

The `autoAssignFloatingIp` option means that a **floating ip** will be assigned to the VM at provision-time.

A floating IP pool name can also be specified. If not set, a floating IP from any available pool will be chosen. This is set using the **template option**. For example:

```
location:
  jclouds:openstack-nova:
    ...
  autoAssignFloatingIp: true
  templateOptions:
    # Pool names to use when allocating a floating IP
    floatingIpPoolNames:
      - "pool name"
```

Basic Location Structure

This is a basic inline **YAML** template for an OpenStack **location**:

```
location:
  jclouds:openstack-nova:
    endpoint: http://x.x.x.x:5000/v2.0/
    identity: "your-tenant:your-username"
    credential: your-password

    # imageId, hardwareId, and loginUser* are optional
    imageId: your-region-name/your-image-id
    hardwareId: your-region-name/your-flavor-id
    loginUser: 'ubuntu'
    loginUser.privateKeyFile: /path/to/your/privatekey

    jclouds.openstack-nova.auto-generate-keypairs: false
    jclouds.openstack-nova.auto-create-floating-ips: true

    templateOptions:
      networks: [ "your-network-id" ]
      floatingIpPoolNames: [ "your-floatingIp-pool" ]
      securityGroups: ['your-security-group']

    # Optional if 'jclouds.openstack-nova.auto-generate-keypairs' is assigned to 'true'
    keyPairName: "your-keypair"
```

This is the same OpenStack **location** in a format that can be added to your `brooklyn.properties` file:

```
brooklyn.location.named.My\ OpenStack=jclouds:openstack-nova:http://x.x.x.x:5000/v2.0/
brooklyn.location.named.My\ OpenStack.identity=your-tenant:your-username
```



```
brooklyn.location.named.My\ OpenStack.credential=your-password
brooklyn.location.named.My\ OpenStack.endpoint=http://x.x.x.x:5000/v2.0/

brooklyn.location.named.My\ OpenStack.imageId=your-region-name/your-image-id
brooklyn.location.named.My\ OpenStack.hardwareId=your-region-name/your-flavor-id
brooklyn.location.named.My\ OpenStack.loginUser=ubuntu
brooklyn.location.named.My\ OpenStack.loginUser.privateKeyFile=/path/to/your/privatekey
brooklyn.location.named.My\ OpenStack.openstack-nova.auto-generate-keypairs=false
brooklyn.location.named.My\ OpenStack.openstack-nova.auto-create-floating-ips=true

brooklyn.location.named.My\ OpenStack.networks=your-network-id
brooklyn.location.named.My\ OpenStack.floatingIpPoolNames=your-floatingip-pool
brooklyn.location.named.My\ OpenStack.securityGroups=your-security-group
brooklyn.location.named.My\ OpenStack.keyPair=your-keypair
```

Troubleshooting

Cloud Credentials Failing

If the cloud API calls return `401 unauthorized` (e.g. in a `org.jclouds.rest.AuthorizationException`), then this could be because the credentials are incorrect.

A good way to check this is to try the same credentials with the [OpenStack nova command line client](#).

Unable to SSH: Wrong User

If SSH authentication fails, it could be that the login user is incorrect. For most clouds, this is inferred from the image metadata, but if no (or the wrong) login user is specified then it will default to root. The correct login user can be specified using the configuration option `loginUser`. For example,

```
loginUser: ubuntu
```

The use of the wrong login user can also result in the obscure message, caused by an unexpected response saying to use a different user. For more technical information, see this [sshj github issue](#). The message is:

```
Received message too long 1349281121
```

I Want to Use My Own KeyPair

By default, jclouds will auto-generate a new [key pair](#) for the VM. This key pair will be deleted automatically when the VM is deleted.

Alternatively, you can use a pre-existing key pair. If so, you must also specify the corresponding private key (pem file, or data) to be used for the initial login. The name used in the `keyPair` configuration must match the name of a key pair that has already been added in OpenStack. For example:

```
location:
  jclouds:clouds:openstack-nova:
    ...
    jclouds.openstack-nova.auto-generate-keypairs: false
    keyPair: "my-keypair"
    loginUser: ubuntu
    loginUser.privateKeyFile: /path/to/my/privatekey.pem
```

Error "doesn't contain ... -----BEGIN"

If using `loginUser.privateKeyFile` (or `loginUser.privateKeyData`), this is expected to be a `.pem` file. If a different format is used (e.g. a `.ppk` file), it will give an error like that below:

```
Error invoking start at EmptySoftwareProcessImpl{id=TrmhitVc}: chars
PuTTY-User-Key-File-2: ssh-rsa
...
doesn't contain % line [-----BEGIN ]
```

Warning Message: "Ignoring request to set..."

If you see a warning log message like that below:

```
2016-06-23 06:05:12,297 WARN o.a.b.l.j.JcloudsLocation [brooklyn-execmanager-X1wkWB3k-312]:
Ignoring request to set template option loginUser because this is not supported by
org.jclouds.openstack.nova.v2_0.compute.options.NovaTemplateOptions
```

It can mean that the `location` configuration option is in the wrong place. The configuration under `templateOptions` must correspond to those options on the `jclouds Nova template options`. However, template options such as `loginUser` are top-level configuration options that should not be inside the `templateOptions` section.

HttpResponseException Accessing Compute Endpoint

The Keystone endpoint is first queried to get the API access endpoints for the appropriate services.

Some private OpenStack installs are (mis)configured such that the returned addresses are not always directly accessible. It could be that the service is behind a VPN, or that they rely on hostnames that are only in a private DNS.

You can find the service endpoints in OpenStack, either using the CLI or the web-console. For example, in Blue Box the URL is https://tenant-region.openstack.blueboxgrid.com/project/access_and_security/. You can then check if the Compute service endpoint is directly reachable.

VM Failing to Provision

It can be useful to drop down to the OpenStack nova CLI, or to jclouds, to confirm that VM provisioning is working and to check which options are required.

For example, try following [these jclouds instructions](#).

jclouds Namespace Issue

A change to Nova's API (in the Mitaka release) resulted in all extensions having the same "fake" namespace which the current version of jclouds does not yet support.

If you are having problems deploying to OpenStack, consult your Brooklyn debug log and look for the following:

```
"namespace": "http://docs.openstack.org/compute/ext/fake_xml"
```

If you already have `jclouds:openstack-mitaka-nova`, then try using this instead of the vanilla `jclouds:openstack-nova`. For example:

```
location:
  jclouds:openstack-mitaka-nova:
    endpoint: http://x.x.x.x:5000/v2.0/
    identity: "your-tenant:your-username"
```

```
credential: your-password
templateOptions:
  networks: [ "your-network-id" ]
  floatingIpPoolNames: [ "your-floatingIp-pool" ]
```

Note that the following values will be set by default when omitted above:

```
jclouds.keystone.credential-type=passwordCredentials
jclouds.openstack-nova.auto-generate-keypairs: true
jclouds.openstack-nova.auto-create-floating-ips: true
```

Inheritance and Named Locations

Named locations can be defined for commonly used groups of properties, with the syntax `brooklyn.location.named.your-group-name.` followed by the relevant properties. These can be accessed at runtime using the syntax `named:your-group-name` as the deployment [location](#).

Some illustrative examples using named locations and showing the syntax and properties above are as follows:

```
# Production pool of machines for my application (deploy to named:prod1)
brooklyn.location.named.prod1=byon:(hosts="10.9.1.1,10.9.1.2,produser2@10.9.2.{10,11,20-29}")
brooklyn.location.named.prod1.user=produser1
brooklyn.location.named.prod1.privateKeyFile=~/.ssh/produser_id_rsa
brooklyn.location.named.prod1.privateKeyPassphrase=s3cr3tCOMPANYpassphrase

# AWS using my company's credentials and image standard, then labelling images so others know they're mine
brooklyn.location.named.company-jungle=jclouds:aws-ec2:us-west-1
brooklyn.location.named.company-jungle.identity=BCDEFGHIJKLMNOPQRSTU
brooklyn.location.named.company-jungle.privateKeyFile=~/.ssh/public_clouds/company_aws_id_rsa
brooklyn.location.named.company-jungle.imageId=ami-12345
brooklyn.location.named.company-jungle.minRam=2048
brooklyn.location.named.company-jungle.userMetadata=application=my-jungle-app,owner="Bob Johnson"
brooklyn.location.named.company-jungle.machineCreateAttempts=2

brooklyn.location.named.AWS\ Virginia\ Large\ Centos = jclouds:aws-ec2
brooklyn.location.named.AWS\ Virginia\ Large\ Centos.region = us-east-1
brooklyn.location.named.AWS\ Virginia\ Large\ Centos.imageId=us-east-1/ami-7d7bfc14
brooklyn.location.named.AWS\ Virginia\ Large\ Centos.user=root
brooklyn.location.named.AWS\ Virginia\ Large\ Centos.minRam=4096
```

Named locations can refer to other named locations using `named:xxx` as their value. These will inherit the configuration and can override selected keys. Properties set in the namespace of the provider (e.g. `b.1.jclouds.aws-ec2.KEY=VALUE`) will be inherited by everything which extends AWS Sub-prefix strings are also inherited up to `brooklyn.location.*`, except that they are filtered for single-word and other known keys (so that we exclude provider-scoped properties when looking at sub-prefix keys). The precedence for configuration defined at different levels is that the value defined in the most specific context will apply.

This is rather straightforward and powerful to use, although it sounds rather more complicated than it is! The examples below should make it clear. You could use the following to install a public key on all provisioned machines, an additional public key in all AWS machines, and no extra public key in `prod1` :

```
brooklyn.location.extraSshPublicKeyUrls=http://me.com/public_key
brooklyn.location.jclouds.aws-ec2.extraSshPublicKeyUrls=[" \"http://me.com/public_key\", \"http://me.com/aws_public_key\" "]
brooklyn.location.named.prod1.extraSshPublicKeyUrls=
```

And in the example below, a config key is repeatedly overridden. Deploying `location: named:my-extended-aws` will result in an `aws-ec2` machine in `us-west-1` (by inheritance) with `VAL6` for `KEY` :

```
brooklyn.location.KEY=VAL1
brooklyn.location.jclouds.KEY=VAL2
brooklyn.location.jclouds.aws-ec2.KEY=VAL3
brooklyn.location.jclouds.aws-ec2@us-west-1.KEY=VAL4
brooklyn.location.named.my-aws=jclouds:aws-ec2:us-west-1
brooklyn.location.named.my-aws.KEY=VAL5
brooklyn.location.named.my-extended-aws=named:my-aws
brooklyn.location.named.my-extended-aws.KEY=VAL6
```

BYON

"Bring-your-own-nodes" mode is useful in production, where machines have been provisioned by someone else, and during testing, to cut down provisioning time.

Your nodes must meet the following prerequisites:

- A suitable OS must have been installed on all nodes
- The node must be running `sshd` (or similar)
- the brooklyn user must be able to ssh to each node as root or as a user with passwordless `sudo` permission. (For more information on SSH keys, see [here](#).)

To deploy to machines with known IP's in a [blueprint](#), use the following syntax:

```
location:
  byon:
    user: brooklyn
    privateKeyFile: ~/.ssh/brooklyn.pem
    hosts:
      - 192.168.0.18
      - 192.168.0.19
```

Some of the login properties as described above for jclouds are supported, but not `loginUser` (as no users are created), and not any of the VM creation parameters such as `minRam` and `imageId` . (These clearly do not apply in the same way, and they are *not* by default treated as constraints, although an [entity](#) can confirm these where needed.) As before, if the brooklyn user and its default key are authorized for the hosts, those fields can be omitted.

Named locations can also be configured in your `brooklyn.properties` , using the format `byon:(key=value,key2=value2)` . For convenience, for hosts wildcard globs are supported.

```
brooklyn.location.named.On-Prem\ Iron\ Example=byon:(hosts="10.9.1.1,10.9.1.2,produser2@10.9.2.{10,11,20-29}")
brooklyn.location.named.On-Prem\ Iron\ Example.user=produser1
brooklyn.location.named.On-Prem\ Iron\ Example.privateKeyFile=~/.ssh/produser_id_rsa
brooklyn.location.named.On-Prem\ Iron\ Example.privateKeyPassphrase=s3cr3tpassphrase
```

Alternatively, you can create a specific BYON [location](#) through the [location](#) wizard tool available within the web console. This [location](#) will be saved as a [catalog entry](#) for easy reusability.

For more complex host configuration, one can define custom config values per machine. In the example below, there will be two machines. The first will be a machine reachable on `ssh -i ~/.ssh/brooklyn.pem -p 8022 myuser@50.51.52.53` . The second is a windows machine, reachable over WinRM. Each machine has also has a private address (e.g. for within a private network).

```
location:
```

```

byon:
  hosts:
    - ssh: 50.51.52.53:8022
      privateAddresses: [10.0.0.1]
      privateKeyFile: ~/.ssh/brooklyn.pem
      user: myuser
    - winrm: 50.51.52.54:8985
      privateAddresses: [10.0.0.2]
      password: mypassword
      user: myuser
      osFamily: windows

```

The BYON `location` also supports a machine chooser, using the config key `byon.machineChooser`. This allows one to plugin logic to choose from the set of available machines in the pool. For example, additional config could be supplied for each machine. This could be used (during the call to `location.obtain()`) to find the config that matches the requirements of the `entity` being provisioned. See `FixedListMachineProvisioningLocation.MACHINE_CHOOSER`.

SSH Keys

SSH keys are one of the simplest and most secure ways to access remote servers. They consist of two parts:

- A private key (e.g. `id_rsa`) which is known only to one party or group
- A public key (e.g. `id_rsa.pub`) which can be given to anyone and everyone, and which can be used to confirm that a party has a private key (or has signed a communication with the private key)

In this way, someone -- such as you -- can have a private key, and can install a public key on a remote machine (in an `authorized_keys` file) for secure automated access. Commands such as `ssh` (and Brooklyn) can log in without revealing the private key to the remote machine, the remote machine can confirm it is you accessing it (if no one else has the private key), and no one snooping on the network can decrypt any of the traffic.

Creating an SSH Key

If you don't have an SSH key, create one with:

```
$ ssh-keygen -t rsa -N "" -f ~/.ssh/id_rsa
```

Localhost Setup

If you want to deploy to `localhost`, ensure that you have a public and private key, and that your key is authorized for ssh access:

```

# _Appends_ id_rsa.pub to authorized_keys. Other keys are unaffected.
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys

```

Now verify that your setup by running the command: `ssh localhost echo hello world`

If your setup is correct, you should see `hello world` printed back at you.

On the first connection, you may see a message similar to this:

```

The authenticity of host 'localhost (::1)' can't be established.
RSA key fingerprint is 7b:e3:8e:c6:5b:2a:05:a1:7c:8a:cf:d1:6a:83:c2:ad.
Are you sure you want to continue connecting (yes/no)?

```

Simply answer 'yes' and then repeat the command again.

If this isn't the case, see below.

Potential Problems

- **MacOS user?** In addition to the above, enable "Remote Login" in "System Preferences > Sharing".
- **Got a passphrase?** Set `brooklyn.location.localhost.privateKeyPassphrase` as described [here](#). If you're not sure, or you don't know what a passphrase is, you can test this by executing `ssh-keygen -y`. If it does *not* ask for a passphrase, then your key has no passphrase. If your key does have a passphrase, you can remove it by running `ssh-keygen -p`.
- Check that you have an `~/.ssh/id_rsa` file (or `id_dsa`) and a corresponding public key with a `.pub` extension; if not, create one as described above
- `~/.ssh/` or files in that directory may have permissions they shouldn't: they should be visible only to the user (apart from public keys), both on the source machine and the target machine. You can verify this with `ls -l ~/.ssh/`: lines should start with `-rw-----` or `-r-----` (or `-rwx-----` for directories). If it does not, execute `chmod go-rwx ~/.ssh ~/.ssh/*`.
- Sometimes machines are configured with different sets of support SSL/TLS versions and ciphers; if command-line `ssh` and `scp` work, but Brooklyn/java does not, check the versions enabled in Java and on both servers.
- Missing entropy: creating and using ssh keys requires randomness available on the servers, usually in `/dev/random`; see [here](#) for more information

Localhost

If passwordless ssh login to `localhost` and passwordless `sudo` is enabled on your machine, you should be able to deploy some blueprints with no special configuration, just by specifying `location: localhost` in [YAML](#).

If you use a passphrase or prefer a different key, these can be configured as follows:

```
location:
  localhost:
    privateKeyFile=~/.ssh/brooklyn_key
    privateKeyPassphrase=s3cr3tPASSPHRASE
```

Alternatively, you can create a specific localhost [location](#) through the [location](#) wizard tool available within the web console. This [location](#) will be saved as a [catalog entry](#) for easy reusability.

Passwordless Sudo

If you encounter issues or for more information, see [SSH Keys Localhost Setup](#).

For some blueprints, passwordless sudo is required. (Try executing `sudo whoami` to see if it prompts for a password. To enable passwordless `sudo` for your account, a line must be added to the system `/etc/sudoers` file.

To edit the file, use the `visudo` command:

```
sudo visudo
```

Add this line at the bottom of the file, replacing `username` with your own user:

```
username ALL=(ALL) NOPASSWD: ALL
```

If executing the following command does not ask for your password, then `sudo` has been setup correctly:

```
sudo whoami
```

Location Customizers

Apache Brooklyn supports a number of ways to configure and customize locations. These include the

`JcloudsLocationCustomizer`, which is for advanced customization of VM provisioning through jclouds. There is also a `MachineLocationCustomizer`, which allows customization of machines being obtained from any kind of [location](#) (including [Bring Your Own Nodes](#)).

Usage Guidelines

Clearly there is an overlap for where things can be done. This section describes the recommended separation of responsibilities.

These are guidelines only - users are obviously free to make alternative usage decisions based on their particular use-cases.

Responsibilities of Entity versus Location

From an [entity](#)'s perspective, it calls `location.obtain(options)` and gets back a usable `MachineLocation` that has a standard base operating system that gives remote access (e.g. for Linux it expects credentials for a user with `sudo` rights, and ssh access).

However, there are special cases - for example the `location.obtain(options)` could return a Docker container with the software pre-installed, and no remote access (see the [Clocker project](#) for more information on use of Docker with Brooklyn).

The [entity](#) is then responsible for configuring that machine according to the needs of the software to be installed.

For example, the [entity](#) may install software packages, upload/update configuration files, launch processes, etc.

The [entity](#) may also configure `iptables`. This is also possible through the `JcloudsLocation` configuration. However, it is preferable to do this in the [entity](#) because it is part of configuring the machine in the way required for the given software component.

The [entity](#) may also perform custom OS setup, such as installing security patches. However, whether this is appropriate depends on the nature of the security patch: if the security patch is specific to the [entity](#) type, then it should be done within the [entity](#); but if it is to harden the base OS to make it comply with an organisation's standards (e.g. to overcome shortcomings of the base image, or to install security patches) then a `MachineLocationCustomizer` is more appropriate.

Location Configuration Options

This refers to standard [location](#) configuration: explicit config keys, and explicit jclouds template configuration that can be passed through.

This kind of configuration is simplest to use. It is the favoured mechanism when it comes to VM provisioning, and should be used wherever possible.

Note that a jclouds `TemplateBuilder` and cloud-specific `TemplateOptions` are the generic mechanisms within jclouds for specifying the details of the compute resource to be provisioned.

Jclouds Location Customizer

A `JcloudsLocationCustomizer` has customization hooks to execute code at the various points of building up the jclouds template and provisioning the machine. Where jclouds is being used and where the required use of jclouds goes beyond simple configuration, this is an appropriate solution.

For example, there is a `org.apache.brooklyn.location.jclouds.networking.JcloudsLocationSecurityGroupCustomizer` which gives more advanced support for setting up security groups (e.g. in AWS-EC2).

Machine Customizer

The `MachineLocationCustomizer` allows customization of machines being obtained from any kind of `location`. For example, this includes for jclouds and for Bring Your Own Nodes (BYON).

It provides customization hooks for when the machine has been provisioned (before it is returned by the `location`) and when the machine is about to be released by the `location`.

An example use would be to register (and de-register) the machine in a CMDB.

Jclouds Location Customizers

Warning: additional methods (i.e. customization hooks) may be added to the `JcloudsLocationCustomizer` interface in future releases. Users are therefore strongly encouraged to sub-class `BasicJcloudsLocationCustomizer`, rather than implementing `JcloudsLocationCustomizer` directly.

The `JcloudsLocationCustomizer` provides customization hooks at various points of the Brooklyn's use of jclouds. These can be used to adjust the configuration, to do additional setup, to do custom logging, etc.

- Customize the `org.jclouds.compute.domain.TemplateBuilder`, before it is used to build the template. This is used to influence the choice of VM image, hardware profile, etc. This hook is not normally required as the `location` configuration options can be used in instead.
- Customize the `org.jclouds.compute.domain.Template`, to be used when creating the machine. This hook is most often used for performing custom actions - for example to create or modify a security group or volume, and to update the template's options to use that.
- Customize the `org.jclouds.compute.options.TemplateOptions` to be used when creating the machine. The `TemplateOptions` could be cast to a cloud-specific sub-type (if this does not have to work across different clouds). Where the use-case is to just set simple configuration on the `TemplateOptions`, consider instead using the config key `templateOptions`, which takes a map of type String to Object - the strings should match the method names in the `TemplateOptions`.
- Customize the `org.apache.brooklyn.location.jclouds.JcloudsMachineLocation` that has been created. For Linux-based VMs, if the config `waitForSshable` was not false, then this machine is guaranteed to be ssh'able. Similarly for WinRM access to Windows machines, if `waitForWinRMAvailable` was not false.
- Pre-release of the machine. If the actions required are specific to jclouds (e.g. using jclouds to make calls to the cloud provider) then this should be used; otherwise one should use the more generic `MachineLocationCustomizer`.

- Post-release of the machine (i.e. after asking jclouds to destroying the machine).

To register a `JcloudsLocationCustomizer` in `YAML`, the config key `customizers` can be used to provide a list of instances. Each instance can be defined using `$brooklyn:object` to indicate the type and its configuration. For example:

```
location:
  jclouds:aws-ec2:us-east-1:
    customizers:
      - $brooklyn:object:
          type: com.acme.brooklyn.MyJcloudsLocationCustomizer
```

To register `JcloudsLocationCustomizer` instances programmatically, set the config key

`JcloudsLocationConfig.JCLOUDS_LOCATION_CUSTOMIZERS` on the `location`, or pass this config option when calling `location.obtain(options)`.

The `SharedLocationSecurityGroupCustomizer` configures a shared security group on Jclouds locations. It only works on AWS and Azure ARM.

To register a `SharedLocationSecurityGroupCustomizer` in `YAML`, you can use the config key `customizers` and configure it with `$brooklyn:object` and `object.fields`. For example:

```
location:
  jclouds:aws-ec2:us-east-1:
    customizers:
      - $brooklyn:object:
          type: org.apache.brooklyn.location.jclouds.networking.SharedLocationSecurityGroupCustomizer
          object.fields: {locationName: "myloc", tcpPortRanges: ["22", "8080", "9443"], udpPortRanges: ["2001", "4013"], cidr: "82.40.153.101/24"}
```

where `cidr` can be optionally set to restrict the range that the ports that are to be opened can be accessed from.

Machine Location Customizers

Warning: additional methods (i.e. customization hooks) may be added to the `MachineLocationCustomizer` interface in future releases. Users are therefore strongly encouraged to sub-class `BasicMachineLocationCustomizer`, rather than implementing `MachineLocationCustomizer` directly.

The `MachineLocationCustomizer` provides customization hooks for when a machine is obtained/released from a `MachineProvisioningLocation`. The following hooks are supported:

- After the machine has been provisioned/allocated, but before it has been returned.
- When the machine is about to be released, but prior to actually destroying/unallocating the machine.

To register a `MachineLocationCustomizer` in `YAML`, the config key `machineCustomizers` can be used to provide a list of instances. Each instance can be defined using `$brooklyn:object` to indicate the type and its configuration. For example:

```
location:
  jclouds:aws-ec2:us-east-1:
    machineCustomizers:
      - $brooklyn:object:
          type: com.acme.brooklyn.MyMachineLocationCustomizer
```

To register `MachineLocationCustomizer` instances programmatically, set the config key

`CloudLocationConfig.MACHINE_LOCATION_CUSTOMIZERS` on the `location`, or pass this config option when calling `location.obtain(options)`.

Hostname Customizer

[org.apache.brooklyn.entity.machine.SetHostnameCustomizer](#) Sets the hostname on an ssh'able machine. Currently only CentOS and RHEL are supported. The customizer can be configured with a hard-coded hostname, or with a freemarker template whose value (after substitutions) will be used for the hostname.

Customizing Cloud Security Groups

Before using [SharedLocationSecurityGroupCustomizer](#), please first refer to [Port Inferencing](#).

A security group is a named collection of network access rules that are use to limit the types of traffic that have access to instances.

Security group is the standard way to set firewall restrictions on the AWS-EC2 environment.

docs.aws.amazon.com/AmazonVPC/latest/UserGuide/VPC_SecurityGroups.html

When deploying to AWS EC2 target, by default Apache Brooklyn creates security group attached to the VM. It is easy to add additional rules to the initial security group using `org.apache.brooklyn.SharedLocationSecurityGroupCustomizer`.

YAML Example:

```
name: ports @ AWS
location: jclouds:aws-ec2:us-west-2:
services:
- type: org.apache.brooklyn.entity.software.base.EmptySoftwareProcess
  brooklyn.config:
    provisioning.properties:
      customizers:
        - $brooklyn:object:
            type: org.apache.brooklyn.location.jclouds.networking.SharedLocationSecurityGroupCustomizer
            object.fields: {tcpPortRanges: ["900-910", "915", "22"], udpPortRanges: ["100", "200-300"], cidr: "82.40.153.101/24"}
```

Make sure that you have rule which makes port 22 accessible from Apache Brooklyn.

Opening ports during runtime.

Apache Brooklyn exposes the [SharedLocationSecurityGroupCustomizer](#) functionality after [entity](#) is deployed just by supplying `effector.add.openInboundPorts: true` "brooklyn.config". Example configuration in [effector](#)

```
location: jclouds:aws-ec2:us-west-2
services:
- type: org.apache.brooklyn.entity.software.base.EmptySoftwareProcess
  brooklyn.config:
    effector.add.openInboundPorts: true
```

Known limitations

Not all cloud providers support Security Group abstraction. `SharedLocationSecurityGroupCustomizer` is known to work well with Amazon EC2.

Other clouds which support Security Groups:

- Openstack
- Azure - jclouds-labs azurecompute implementation uses endpoints rules when creating a VM instance. jclouds:azurecompute based [location](#) do not have security groups so [SharedLocationSecurityGroupCustomizer](#) is

used it will fail to find a security group.

Specialized Locations

Some additional `location` types are supported for specialized situations:

Single Host

The spec `host`, taking a string argument (the address) or a map (`host` , `user` , `password` , etc.), provides a convenient syntax when specifying a single host. For example:

```
location: host:(192.168.0.1)
services:
- type: org.apache.brooklyn.entity.webapp.jboss.JBoss7Server
```

Or, in `brooklyn.properties`, set `brooklyn.location.named.host1=host:(192.168.0.1)`.

The Multi Location

The spec `multi` allows multiple locations, specified as `targets`, to be combined and treated as one `location`.

Sequential Consumption

In its simplest form, this will use the first target `location` where possible, and will then switch to the second and subsequent locations when there are no machines available.

In the example below, it provisions the first node to `192.168.0.1`, then it provisions into AWS us-east-1 region (because the bring-your-own-nodes region will have run out of nodes).

```
location:
  multi:
    targets:
      - byon:(hosts=192.168.0.1)
      - jclouds:aws-ec2:us-east-1
services:
- type: org.apache.brooklyn.entity.group.DynamicCluster
  brooklyn.config:
    cluster.initial.size: 3
    dynamiccluster.memberspec:
      $brooklyn:entitySpec:
        type: org.apache.brooklyn.entity.machine.MachineEntity
```

Round-Robin Consumption and Availability Zones for Clustered Applications

A `dynamiccluster` can be configured to cycle through its deployment targets round-robin when provided with a `location` that supports the `AvailabilityZoneExtension` -- the `multi` `location` supports this extension.

The configuration option `dynamiccluster.zone.enable` on `dynamiccluster` tells it to query the given `location` for `AvailabilityZoneExtension` support. If the `location` supports it, then the cluster will query for the list of availability zones (which in this case is simply the list of targets) and deploy to them round-robin.

In the example below, the cluster will request VMs round-robin across three different locations (in this case, the locations were already added to the catalog, or defined in `brooklyn.properties`).

```
location:
  multi:
    targets:
      - my-location-1
      - my-location-2
      - my-location-3
services:
- type: org.apache.brooklyn.entity.group.DynamicCluster
  brooklyn.config:
    dynamiccluster.zone.enable: true
    cluster.initial.size: 3
    dynamiccluster.memberspec:
      $brooklyn:entitySpec:
        type: org.apache.brooklyn.entity.machine.MachineEntity
```

Of course, clusters can also be deployed round-robin to real availability zones offered by cloud providers, as long as their locations support `AvailabilityZoneExtension` . Currently, only AWS EC2 locations support this feature.

In the example below, the cluster will request VMs round-robin across the availability zones provided by AWS EC2 in the "us-east-1" region.

```
location: jclouds:aws-ec2:us-east-1
services:
- type: org.apache.brooklyn.entity.group.DynamicCluster
  brooklyn.config:
    dynamiccluster.zone.enable: true
    cluster.initial.size: 3
    dynamiccluster.memberspec:
      $brooklyn:entitySpec:
        type: org.apache.brooklyn.entity.machine.MachineEntity
```

For more information about AWS EC2 availability zones, see [this guide](#).

Custom alternatives to round-robin are also possible using the configuration option

```
dynamiccluster.zone.placementStrategy ON DynamicCluster .
```

The Server Pool

The `ServerPool` entity type allows defining an `entity` which becomes available as a `location`.

Provisioned Machine Requirements

The requirements for how a provisioned machine should behave will depend on the entities subsequently deployed there.

Below are a set of common assumptions, made by many [entity](#) implementations, which could cause subsequent errors if they do not hold. These relate to the machine's configuration, rather than additional networking or security that a given Cloud might offer.

Also see the [Troubleshooting](#) docs.

Remote Access

SSH or WinRM Access

Many entities require ssh'ing (or using WinRM for Windows), to install and configure the software.

An example of disabling all ssh'ing is shown below:

```
location:
  aws-ec2:us-east-1:
    identity: XXXXXXXX
    credential: XXXXXXXX
    waitForSshable: false
    pollForFirstReachableAddress: false
services:
- type: org.apache.brooklyn.entity.software.base.EmptySoftwareProcess
  brooklyn.config:
    onbox.base.dir.skipResolution: true
    sshMonitoring.enabled: false
```

Parsing SSH stdout: No Extra Lines

For entities that execute ssh commands, these sometimes parse the resulting stdout.

It is strongly recommended that VMs are configured so that no additional stdout is written when executing remote ssh (or WinRM) commands. Such stdout risks interfering with the response parsing in some blueprints.

For example, if configuring the VM to write out "Last login" information, this should be done for only "interactive" shells (see [Stackoverflow](#) for more details).

Passwordless Sudo

Does passwordless sudo work?

Try executing:

```
sudo whoami
```

See [Passwordless Sudo](#).

Advertised Addresses

Hostname Resolves Locally

Does the hostname known at the box resolve at the box?

Try executing:

```
ping $(hostname)
```

if not, consider setting `generate.hostname: true` in the `location` config, for jclouds-based locations.

IP Resolves Locally

For the IP address advertised in Brooklyn using the `sensor` `host.addresses.private` (or `host.subnet.address`), can the machine reach that IP?

Get the `sensor` value, and then try executing:

```
ping ${PRIVATE_IP}
```

Is there a public IP (advertised using the `sensor` `host.addresses.public`, or `host.address`), and can the machine reach it?

Get the `sensor` value, and then try executing:

```
ping ${PUBLIC_IP}
```

Networking

Public Internet Access

Can the machine reach the public internet, and does DNS resolve?

Try executing:

```
ping www.example.org
```

Machine's Hostname in DNS

Is the machine hostname well-known? If ones does a DNS lookup, e.g. from the Brooklyn server, does it resolve and does it return the expected IP (e.g. the same IP as the `host.addresses.public` `sensor`)? Try using the hostname that the machine reports when you execute `hostname`.

Many blueprints do not require this, instead using IP addresses directly. Some blueprints may include registration with an appropriate DNS server. Some clouds do this automatically.

Reachability

When provisioning two machines, can these two machines reach each other on the expected IP(s) and hostname(s)?

Try using `ping` from one machine to another using the public or subnet ip or hostname. However, note that `ping` requires access over ICMP, which may be disabled. Alternatively, try connecting to a specific TCP port using `telnet <address> <port>` .

Firewalls

What firewall(s) are running on the machine, and are the required ports open? On linux, check things like `iptables` , `firewalld` , `ufw` or other commercial firewalls. On Windows, check the settings of the [Windows Firewall](#).

Consider using `openIptables: true` , or even `stopIptables: true` .

Sufficient Entropy for /dev/random

Is there sufficient entropy on the machine, for `/dev/random` to respond quickly?

Try executing:

```
{ cat /dev/random > /tmp/x & } ; sleep 10 ; kill %1 ; { cat /dev/random > /tmp/x & } ; sleep 1 ; kill %1 ; wc /tmp/x | awk '{print $3}'
```

The result should be more than 1M.

If not, consider setting `installDevUrandom: true` for jclouds-based locations.

See instructions to [Increase Entropy](#).

File System

Permissions of /tmp

Is `/tmp` writable?

Try executing:

```
touch /tmp/amp-test-file ; rm /tmp/amp-test-file
```

Are files in `/tmp` executable (e.g. some places it has been mounted NO_EXECUTE)?

Try executing:

```
echo date > /tmp/brooklyn-test.sh && chmod +x /tmp/brooklyn-test.sh && /tmp/brooklyn-test.sh && rm /tmp/brooklyn-test.sh
```

Reference Guide

- [Production Installation](#)
- [Starting, Stopping and Monitoring](#)
- [Server CLI Reference](#)
- [Client CLI Reference](#)
- [GUI Guide](#)
- [REST API](#)
- [Configuring Brooklyn](#)
- [Persistence](#)
- [High Availability](#)
- [Logging](#)
- [Externalized Configuration](#)
- [Requirements](#)
- [Upgrade](#)
- [Security Guidelines](#)
- [Troubleshooting](#)
- [Paths breakdown](#)

Production Installation

To install Apache Brooklyn on a production server:

1. [Set up the prerequisites](#)
2. [Download Apache Brooklyn](#)
3. [Configuring brooklyn.cfg](#)
4. [Configuring Karaf Security](#)
5. [Configuring default.catalog.bom](#)
6. [Test the installation](#)

This guide covers the basics. You may also wish to configure:

- [Logging](#)
- [Persistence](#)
- [High availability](#)

Set up the Prerequisites

Check that the server meets the [requirements](#). Then configure the server as follows:

- install Java JRE or JDK (version 8 or later)
- enable "Java Cryptography Extension" (already enabled out of the box of OpenJDK installs)
- install an [SSH key](#), if not available
- if the "localhost" [location](#) will be used, enable [passwordless ssh login](#)
- create a `~/.brooklyn` directory on the host with `$ mkdir ~/.brooklyn`
- check your `iptables` or other firewall service, making sure that incoming connections on port 8443 is not blocked
- check that the [linux kernel entropy](#) is sufficient
- check that the [ulimit values](#) are sufficiently high
- ensure external libraries are up-to-date, including `nss` for SSL.
- ensure the time is continually accurate, ideally by running a service like the [ntp daemon](#).

Download Apache Brooklyn

Download Brooklyn and obtain a binary build as described on [the download page](#).

Expand the `tar.gz` archive (note: as this is a -SNAPSHOT version, your filename will be slightly different):

```
tar -zxf apache-brooklyn-dist-0.12.0-timestamp-dist.tar.gz
```

This will create a `apache-brooklyn-0.12.0` folder.

Let's setup some paths for easy commands.

```
% cd apache-brooklyn-0.12.0
% BROOKLYN_DIR="$(pwd)"
% export PATH=$PATH:$BROOKLYN_DIR/bin/
```

Configuring brooklyn.cfg

Set up `brooklyn.cfg` as described [here](#):

- Configure the users who should have access
- Turn on HTTPS
- Supply credentials for any pre-defined clouds

Configuring Karaf Security

Out of the box, Apache Brooklyn includes the default Karaf security configuration. This configuration is used to manage connections to the ssh port of Karaf (which is available to localhost connections only). It is recommended that you update the credentials as detailed in the [Karaf Security](#) page.

Configuring the Catalog

By default Brooklyn loads the catalog of available application components and services from `default.catalog.bom` on the classpath. The initial catalog is in `conf/brooklyn/` in the dist. If you have a preferred catalog, simply replace that file.

[More information on the catalog is available here.](#)

Confirm Installation

Launch Brooklyn in a disconnected session so it will remain running after you have logged out:

```
% nohup bin/brooklyn launch > /dev/null 2&>1 &
```

Apache Brooklyn should now be running on port 8081 (or other port if so specified).

To install on a different port edit config in `etc/org.ops4j.pax.web.cfg`.

Starting, Stopping and Monitoring

NOTE: This document is for information on starting an Apache Brooklyn Server. For information on using the Brooklyn Client CLI to access an already running Brooklyn Server, refer to [Client CLI Reference](#).

Packages for RHEL/CentOS and Ubuntu

If you are using the `.rpm` or `.deb` package of Apache Brooklyn, then Brooklyn will integrate with your OS service management. Commands such as `service brooklyn start` will work as expected, and Brooklyn's PID file will be stored in the normal [location](#) for your OS, such as `/var/run/brooklyn.pid`.

Platform-independent distributions

The platform-independent distributions are packaged in `.tar.gz` and `.zip` files.

Starting

To launch Brooklyn, from the directory where Brooklyn is unpacked, run:

```
% bin/start
```

With no configuration, this will launch the Brooklyn web console and REST API on `http://localhost:8081/`, listening on all network interfaces. No credentials are required by default. It is strongly recommended to [configure security](#).

See the [Server CLI Reference](#) for more information about the Brooklyn server process.

Stopping

To stop Brooklyn, from the directory where Brooklyn is unpacked, run:

For example:

```
% bin/stop
```

Monitoring

For `.tar.gz` and `.zip` distributions of Brooklyn, the Brooklyn startup script will create a file name `pid_java` at the root of the Brooklyn directory, which contains the PID of the last Brooklyn process to be started. You can examine this file to discover the PID, and then test that the process is still running. `.rpm` and `.deb` distributions of Brooklyn will use the normal mechanism that your OS uses, such as writing to `/var/run/brooklyn.pid`.

This should lead to a fairly straightforward integration with many monitoring tools - the monitoring tool can discover the expected PID, and can execute the start or stop commands shown above as necessary.

For example, here is a fragment of a `monitrc` file as used by [Monit](#), for a Brooklyn `.tar.gz` distribution unpacked and installed at `/opt/apache-brooklyn`:

```
check process apachebrooklyn with pidfile /opt/apache-brooklyn/pid_java
    start program = "/bin/bash -c '/opt/apache-brooklyn/bin/brooklyn launch --persist auto & disown'" with time
    out 10 seconds
    stop program = "/bin/bash -c 'kill $( cat /opt/apache-brooklyn/pid_java )'"
```

In addition to monitoring the Brooklyn process itself, you will almost certainly want to monitor resource usage of Brooklyn. In particular, please see the [Requirements](#) section for a discussion on Brooklyn's disk space requirements.

Server CLI Reference

NOTE: This document is for information on starting a Brooklyn Server. For information on using the Brooklyn Client CLI to access an already running Brooklyn Server, refer to [Client CLI Reference](#).

Launch command

To launch Brooklyn, from the directory where Brooklyn is unpacked, run:

```
% nohup bin/brooklyn launch > /dev/null 2>&1 &
```

With no configuration, this will launch the Brooklyn web console and REST API on <http://localhost:8081/>, listening on all network interfaces. No credentials are required by default. For a production system, or if Apache Brooklyn is publicly reachable, it is strongly recommended to [configure security](#).

By default, Brooklyn will write log messages at the INFO level or above to `brooklyn.info.log` and messages at the DEBUG level or above to `brooklyn.debug.log`. Redirecting the output to `/dev/null` prevents the default console output being written to `nohup.out`.

You may wish to [add Brooklyn to your path](#); assuming you've done this, to get information the supported CLI options at any time, just run `brooklyn help`:

```
% bin/brooklyn help

usage: brooklyn [(-q | --quiet)] [(-v | --verbose)] <command> [<args>]

The most commonly used brooklyn commands are:
  help      Display help information about brooklyn
  info      Display information about brooklyn
  launch     Starts a brooklyn application. Note that a BROOKLYN_CLASSPATH environment variable needs to be set
            up beforehand to point to the user application classpath.

See 'brooklyn help <command>' for more information on a specific command.
```

It is important that Brooklyn is launched with either `nohup ... &` or `... & disown`, to ensure it keeps running after the shell terminates.

Other Server CLI Arguments

The Server CLI arguments for [persistence and HA](#) and the [catalog](#) are described separately.

Path Setup

In order to have easy access to the server cli it is useful to configure the PATH environment variable to also point to the cli's bin directory:

```
BROOKLYN_HOME=/path/to/brooklyn/
export PATH=$PATH:$BROOKLYN_HOME/usage/dist/target/brooklyn-dist/bin/
```

Memory Usage

The amount of memory required by the Brooklyn process depends on the usage -- for example the number of entities/VMs under management.

For a standard Brooklyn deployment, the defaults are to start with 256m, and to grow to 1g of memory. These numbers can be overridden by setting the environment variable `JAVA_OPTS` before launching the `brooklyn script`, as follows:

```
JAVA_OPTS="-Xms1g -Xmx4g"
```

(On Java 8 and later the last entry has no effect and can be dropped.)

Brooklyn stores a task history in-memory using [soft references](#). This means that, once the task history is large, Brooklyn will continually use the maximum allocated memory. It will only expunge tasks from memory when this space is required for other objects within the Brooklyn process.

See [Memory Usage](#) for more information on memory usage and other suggested `JAVA_OPTS`.

Web Console Bind Address

The web console will by default bind to 0.0.0.0. It's restricted to 127.0.0.1 if the `--noconsoleSecurity` flag is used. To specify a local interface, or use the local loopback (127.0.0.1), for the web console to bind to you should use:

```
--bindAddress <IP>
```

Configuration

Configuration Files

Brooklyn reads configuration from a variety of places. It aggregates the configuration. The list below shows increasing precedence (i.e. the later ones will override values from earlier ones, if exactly the same property is specified multiple times).

1. `classpath://brooklyn/location-metadata.properties` is shipped as part of Brooklyn, containing generic metadata such as jurisdiction and geographic information about Cloud providers.
2. The file `~/brooklyn/location-metadata.properties` (unless `--noGlobalBrooklynProperties` is specified). This is intended to contain custom metadata about additional locations.
3. The file `brooklyn.cfg` (unless `--noGlobalBrooklynProperties` is specified).
4. Another properties file, if the `--localBrooklynProperties <local brooklyn.properties file>` is specified.
5. Shell environment variables
6. System properties, supplied with `-D` on the brooklyn (Java) command-line.

These properties are described in more detail [here](#).

Extending the Classpath

The default Brooklyn directory structure includes:

- `./conf/` : for configuration resources.
- `./lib/patch/` : for Jar files containing patches.
- `./lib/brooklyn/` : for the brooklyn libraries.
- `./lib/dropins/` : for additional Jars.

Resources added to `conf/` will be available on the classpath.

A patch can be applied by adding a Jar to the `lib/patch/` directory, and restarting Brooklyn. All jars in this directory will be at the head of the classpath.

Additional Jars should be added to `lib/dropins/`, prior to starting Brooklyn. These jars will be at the end of the classpath.

The initial classpath, as set in the `brooklyn` script, is:

```
conf:lib/patch/*:lib/brooklyn/*:lib/dropins/*
```

Additional entries can be added at the head of the classpath by setting the environment variable `BROOKLYN_CLASSPATH` before running the `brooklyn` script.

Replacing the web-console

Work in progress.

The Brooklyn web-console is loaded from the classpath as the resource `classpath:///brooklyn.war`.

To replace this, an alternative WAR with that name can be added at the head of the classpath. However, this approach is likely to change in a future release - consider this feature as "beta".

Cloud Explorer

The `brooklyn` command line tool includes support for querying (and managing) cloud compute resources and blob-store resources.

For example, `brooklyn cloud-compute list-instances --location aws-ec2:eu-west-1` will use the AWS credentials from `brooklyn.properties` and list the VM instances running in the given EC2 region.

Use `brooklyn help` and `brooklyn help cloud-compute` to find out more information.

This functionality is not intended as a generic cloud management CLI, but instead solves specific Brooklyn use-cases.

The main use-case is discovering the valid configuration options on a given cloud, such as for `imageId` and `hardwareId`.

Cloud Compute

The command `brooklyn cloud-compute` has the following options:

- `list-images` : lists VM images within the given cloud, which can be chosen when provisioning new VMs. This is useful for finding the possible values for the `imageId` configuration.
- `get-image <imageId1> <imageId2> ...` : retrieves metadata about the specific images.
- `list-hardware-profiles` : lists the ids and the details of the hardware profiles available when provisioning. This is useful for finding the possible values for the `hardwareId` configuration.
- `default-template` : retrieves metadata about the image and hardware profile that will be used by Brooklyn for that `location`, if no additional configuration options are supplied.
- `list-instances` : lists the VM instances within the given cloud.
- `terminate-instances <instanceId1> <instanceId2> ...` : Terminates the instances with the given ids.

Blob Store

The command `brooklyn cloud-blobstore` is used to access a given object store, such as S3 or Swift. It has the following options:

- `list-containers` : lists the containers (i.e. buckets in S3 terminology) within the given object store.
- `list-container <containerName>` : lists all the blobs (i.e. objects) contained within the given container.
- `blob --container <containerName> --blob <blobName>` : retrieves the given blob (i.e. object), including metadata and its contents.

NOTE: These documents are for using the Brooklyn Client CLI tool to access a running Brooklyn Server. For information on starting on a Brooklyn Server, refer to [Server CLI Reference](#).

Obtaining the CLI tool

A selection of distributions of the CLI tool, `br`, are available to download from the download site [here](#).

Alternatively the CLI tool is available as an executable binary for many more platforms in the Apache Brooklyn distribution, under `bin/brooklyn-client-cli/`, with each build in its own subdirectory:

- Mac: `darwin.amd64/`
- Windows 32-bit: `windows.386/`
- Windows 64-bit: `windows.amd64/`
- Linux 32-bit: `linux.386/`
- Linux 64-bit: `linux.amd64/`

The binary is completely self-contained so you can either copy it to your `bin/` directory or add the appropriate directory above to your path:

```
PATH=$PATH:$HOME/apache-brooklyn/bin/brooklyn-client-cli/linux.amd64/
```

Documentation

- [CLI Reference Guide](#)
- [CLI Usage Guide](#)

CLI Reference Guide

Usage

```
NAME:
  br - A Brooklyn command line client application

USAGE:
  br [global options] command [command options] [arguments...]
```

List of Commands

Commands whose description begins with a `*` character are particularly experimental and likely to change in upcoming releases.

```
COMMANDS:

  Commands whose description begins with a "*" character are particularly experimental and likely to change in
  upcoming
  releases.  If not otherwise specified, "SCOPE" below means application or entity scope.  If an entity scope
  is not
  specified, the application entity is used as a default.

  access          Show access control
  activity        Show the activity for an application / entity
  add-catalog      (deprecated, use catalog add) Add a new catalog item from the supplied YAML (a file or ht
  tp URL)
  add-children     * Add a child or children to this entity from the supplied YAML
  application      Show the status and location of running applications
  catalog          Catalog operations
  config           Show the config for an application or entity
  delete           * Delete (expunge) a brooklyn application
  deploy           Deploy a new application from the given YAML (read from file or URL, or stdin)
  destroy-policy   Destroy a policy
  effector         Show the effectors for an application or entity
  entity           Show the entities of an application or entity
  env              Show the ENV stream for a given activity
  invoke           Invoke an effector of an application and entity
  locations        * List the available locations
  login            Login to brooklyn
  policy           Show the policies for an application or entity
  rename           Rename an application or entity
  restart          Invoke restart effector on an application and entity
  sensor           Show values of all sensors or named sensor for an application or entity
  set              Set config for an entity
  spec            Get the YAML spec used to create the entity, if available
  start            Invoke start effector on an application and entity
  start-policy     Start or resume a policy
  stderr           Show the STDERR stream for a given activity
  stdin            Show the STDIN stream for a given activity
  stdout           Show the STDOUT stream for a given activity
  stop             Invoke stop effector on an application and entity
  stop-policy      Suspends a policy
  tree            * Show the tree of all applications
  version          Display the version of the connected Brooklyn
  help
```

```
GLOBAL OPTIONS:
```

```

--skipSslChecks    Skip verification of server's certificate chain and hostname (for use with self-signed certificates)
--help, -h        show help
--version, -v      print the version

```

Scopes

Many commands require a "scope" expression to indicate the target on which they operate. Where this is required the usage statements below will use the shorthand nomenclature of `<X-scope>`.

The various scopes should be replaced on the command line as:

- `<app-scope>`
application `<Name|AppID>`
- `<entity-scope>`
application `<Name|AppID>` entity `<Name|EntityID>`
- `<effector-scope>`
application `<Name|AppID>` effector `<Name>`
application `<Name|AppID>` entity `<Name|EntityID>` effector `<Name>`
- `<config-scope>`
application `<Name|AppID>` entity `<Name|EntityID>` config `<ConfigID>`
- `<activity-scope>`
activity `<ActivityID>`
application `<Name|AppID>` entity `<Name|EntityID>` activity `<ActivityID>`

Abbreviations

Many of the commands and scopes have shortened aliases:

```

activity    act
application app
entity      ent
policy      pol

```

Command Reference

Login

- `br login <URL> [username [password]]`
Login to Brooklyn. The CLI will prompt for a password if it is not provided. If the Brooklyn server is running on localhost with no security enabled, the username and password may be omitted.
On successful login, the version of the connected Brooklyn server is shown.
- `br version` Show the version of the connected Brooklyn server.

Applications

- `br deploy (<FILE|URL> | -)`
Deploy an application based on the supplied **YAML** file or URL, or read from STDIN when `-` is given instead of a file name.

- `br application`
List the running applications.
- `br application <Name|AppID>`
Show the detail for an application.
- `br <app-scope> config`
Show the configuration details for an application.
- `br <app-scope> config <ConfigID>`
Show the value for a configuration item.
- `br <app-scope> spec`
Show the **YAML** specification used to create the application.
- `br <app-scope> rename <Name>`
Rename the application to .
- `br <app-scope> stop`
Stop an application. See below for further information on the `stop` **effector**.
- `br <app-scope> start`
Start an application. See below for further information on the `start` **effector**.
- `br <app-scope> restart`
Restart an application. See below for further information on the `restart` **effector**.
- `br <app-scope> delete`
Delete an application from Brooklyn.
NOTE: Use this command with care. Even if the application / entities are still running, Brooklyn will drop all knowledge of them and they will be left running in an 'orphaned' state.

Entities

- `br <app-scope> entity`
List the child entities for an application.
- `br <entity-scope> entity`
List the child entities for an **entity**.
- `br <app-scope> entity <Name|EntityID>`
Show the detail of an **entity**.
- `br <app-scope> entity -c <Name|EntityID>`
List the child entities for an **entity**.
- `br <entity-scope> config`
Show the configuration details for an **entity**.
- `br <entity-scope> config <ConfigID>`
Show the value for a configuration item.
- `br <config-scope> set <ConfigValue>`
Set the value of a configuration item.
- `br <entity-scope> spec`
Show the **YAML** specification used to create the **entity**.

- `br <entity-scope> rename <Name>`
Rename the `entity` to .
- `br <entity-scope> stop`
Stop an `entity`. See below for further information on the `stop` `effector`.
- `br <entity-scope> start`
Start an `entity`. See below for further information on the `start` `effector`.
- `br <entity-scope> restart`
Restart an `entity`. See below for further information on the `restart` `effector`.

Sensors

- `br <app-scope> sensor`
List the sensors and values for an application.
- `br <app-scope> sensor <SensorID>`
Show the value for a `sensor`.
- `br <entity-scope> sensor`
List the sensors and values for an `entity`.
- `br <entity-scope> sensor <SensorID>`
Show the value for a `sensor`.

Effectors

- `br <app-scope> effector`
List the effectors for an application.
- `br <app-scope> effector <EffectorID>`
Show the detail for an application `effector`.
- `br <app-scope> effector <EffectorID> invoke`
Invoke the `effector` without any parameters.
- `br <app-scope> effector <EffectorID> invoke [-P <param>=<value> ...]` Invoke the `effector` with one of more parameters.
- `br <entity-scope> effector`
List the effectors for an `entity`.
- `br <entity-scope> effector <EffectorID>`
Show the detail for an `entity` `effector`.
- `br <entity-scope> effector <EffectorID> invoke`
Invoke the `effector` without any parameters.
- `br <entity-scope> effector <EffectorID> invoke [-P <param>=<value> ...]` Invoke the `effector` with one of more parameters.
If the parameter value is complex or multi-lined it may be provided in a file and referenced as:
`<param>=@<FILE>`

NOTE Shortcut commands have been provided for the standard start, restart and stop effectors. For example:

- `br <app-scope> stop`
- `br <entity-scope> restart restartChildren=true`

Policies

- `br <entity-scope> policy`
List the policies for an `entity`.
- `br <entity-scope> policy <PolicyID>`
Show the detail for an `entity policy`.
- `br <entity-scope> start-policy <PolicyID>`
Start an `entity policy`.
- `br <entity-scope> stop-policy <PolicyID>`
Stop an `entity policy`.
- `br <entity-scope> destroy-policy <PolicyID>`
Destroy an `entity policy`.

Activities

- `br <app-scope> activity`
List the activities for an application.
- `br <entity-scope> activity`
List the activities for an `entity`.
- `br <activity-scope> activity`
List the activities for an activity (ie its children).
- `br activity <ActivityID>`
Show the detail for an activity.
- `br activity -c <ActivityID>`
List the child activities of an activity.
- `br <activity-scope> stdin`
Show the `<STDIN>` stream for an activity.
- `br <activity-scope> stdout`
Show the `<STDOUT>` stream for an activity.
- `br <activity-scope> stderr`
Show the `<STDERR>` stream for an activity.
- `br <activity-scope> env`
Show the Environment for an activity.

Miscellaneous

Applications

- `br tree`
List all of the applications and entities in a tree representation.

Entities

- `br <entity-scope> add-children <FILE|URL>`
Add a child or children to the `entity` from local `YAML` file or a URL.

Catalog

- `br catalog list <TYPE>`
List the application catalog, where `TYPE` is one of "application", "`entity`", "`location`", or "`policy`"
- `br catalog delete <TYPE> <ITEM_ID:VERSION>`
Delete an item from the catalog, where `TYPE` is as above, and supplying the item's id and version
- `br catalog add <FILE|URL>`
Add catalog entries from a local `YAML` file or a URL. The id and version of added entries are displayed.
- `br locations`
List the `location` catalog. (Includes all locations including those defined in `brooklyn.cfg`)
- `br access`
Show if you have access to provision locations.

CLI Usage Guide

This document provides a brief overview of using the most common Brooklyn CLI commands, by using the CLI to deploy an application then examine various aspects of it.

The [YAML blueprint](#) for the application that will be deployed is shown at the end of this document.

NOTE: In the sample output, some additional line-wrapping has been used to aid readability. Additionally, the vertical bar character is omitted from table output for readability. For scripting purposes it can be useful in conjunction with a shell pipeline like `some_command | cut -f 1 -d '|' | xargs -L1 some_other_command`.

Login

First, login to the running Brooklyn server. This example assumes that the Brooklyn server is running on `localhost`; change the URL and credentials as necessary.

```
$ br login http://localhost:8081 admin
Enter Password: *
Connected to Brooklyn version 0.9.0-SNAPSHOT at http://localhost:8081
```

The version of the connected Brooklyn server may be viewed with the `version` command:

```
$ br version
0.9.0-SNAPSHOT
```

Applications

Deploy the application; on success the Id of the new application is displayed:

```
$ br deploy webapp-policy.yaml
Id:      lm0cZbsT
Name:    WebCluster
Status:  In progress
```

The `application` command can be used to list a summary of all the running applications. After all of the entities have been started, the application status changes to `RUNNING`:

```
$ br application
Id      Name      Status  Location
YeEQHwgW AppCluster RUNNING CNTBotjI
lm0cZbsT WebCluster RUNNING CNTBotjI
```

Further details of an application can be seen by using the ApplicationID or Name as a parameter for the `application` command:

```
$ br application WebCluster
Id:      lm0cZbsT
Name:    WebCluster
Status:  RUNNING
ServiceUp: true
Type:    org.apache.brooklyn.entity.stock.BasicApplication
```



```

CatalogItemId: null
LocationId: CNTB0tjI
LocationName: FixedListMachineProvisioningLocation:CNTB
LocationSpec: byon
LocationType: org.apache.brooklyn.location.byon.FixedListMachineProvisioningLocation

```

The configuration details of an application can be seen with the `config` command:

```

$ br application WebCluster config
Key          Value
camp.template.id TYWVroRz
brooklyn.wrapper_app true

```

Entities

The entities of an application can be viewed with the `entity` command:

```

$ br app WebCluster entity
Id      Name      Type
x0cMooka WebApp    org.apache.brooklyn.entity.webapp.ControlledDynamicWebAppCluster
thHnLFkP WebDB     org.apache.brooklyn.entity.database.mysql.MySqlNode

```

It is common for an **entity** to have child entities; these can be listed by providing an **entity**-scope for the `entity` command:

```

$ br app WebCluster entity WebApp entity
Id      Name      Type
e5pWaiHf Cluster of TomcatServer org.apache.brooklyn.entity.webapp.DynamicWebAppCluster
CZ8QUVgX NginxController:CZ8Q   org.apache.brooklyn.entity.proxy.nginx.NginxController

```

or by using `-c` (or `--children`) flag with the `entity` command:

```

$ br app WebCluster entity -c e5pWaiHf
Id      Name      Type
x0P2LRxZ quarantine org.apache.brooklyn.entity.group.QuarantineGroup
QK6QjmrW TomcatServer:QK6Q org.apache.brooklyn.entity.webapp.tomcat.TomcatServer

```

As for applications, the configuration details of an **entity** can be seen with the `config` command:

```

$ br app WebCluster entity thHnLFkP config
Key          Value
install.unique_label MySqlNode_5.6.26
brooklyn.wrapper_app true
datastore.creation.script.url https://bit.ly/brooklyn-visitors-creation-script
camp.template.id dnw3GqN0
camp.plan.id db
onbox.base.dir /home/vagrant/brooklyn-managed-processes
onbox.base.dir.resolved true

```

The value of a single configuration item can be displayed by using the configuration key as a parameter for the `config` command:

```

$ br app WebCluster entity thHnLFkP config datastore.creation.script.url
https://bit.ly/brooklyn-visitors-creation-script

```

The value of a configuration item can be changed by using the `set` command:

```
$ br app WebCluster entity thHnLFkP config datastore.creation.script.url set \"https://bit.ly/new-script\"
```

Sensors

The sensors associated with an application or `entity` can be listed with the `sensor` command:

```
$ br app WebCluster entity CZ8QUVgX sensor
Name                                     Value
download.addon.urls:                   {"stickymodule":"https://bitbucket.org/nginx-goodies/n
                                         ginx-sticky-module-ng/get/${addonversion}.tar.gz","pcr
                                         e":"ftp://ftp.csx.cam.ac.uk/pub/software/programming/p
                                         cre/pcrc-${addonversion}.tar.gz"}
download.url:                           http://nginx.org/download/nginx-${version}.tar.gz
expandedinstall.dir:                   /home/vagrant/brooklyn-managed-processes/installs/Ngi
                                         nxController_1.8.0/nginx-1.8.0
host.address:                           192.168.52.102
host.name:                             192.168.52.102
host.sshAddress:                        vagrant@192.168.52.102:22
host.subnet.address:                   192.168.52.102
host.subnet.hostname:                  192.168.52.102
http.port:                             8000
install.dir:                           /home/vagrant/brooklyn-managed-processes/installs/Ngin
                                         xController_1.8.0
log.location:                          /home/vagrant/brooklyn-managed-processes/apps/FoEXxwJ2
                                         /entities/NginxController_CZ8QUVgX/console
main.uri:                              http://192.168.52.102:8000/
member.sensor.hostandport:
member.sensor.hostname:                {"typeToken":null,"type":"java.lang.String","name":"ho
                                         st.subnet.hostname","description":"Host name as known
                                         internally in the subnet where it is running (if diffe
                                         rent to host.name)","persistence":"REQUIRED"}
member.sensor.portNumber:              {"typeToken":null,"type":"java.lang.Integer","name":"h
                                         tp.port","description":"HTTP port","persistence":"RE
                                         QUIRED","configKey":{"name":"http.port","typeToken":nu
                                         ll,"type":"org.apache.brooklyn.api.location.PortRange"
                                         ,"description":"HTTP port","defaultValue":{"ranges":[{"
                                         port":8080}, {"start":18080,"end":65535,"delta":1}]},"
                                         reconfigurable":false,"inheritance":null,"constraint":
                                         "ALWAYS_TRUE"}}
nginx.log.access:                      /home/vagrant/brooklyn-managed-processes/apps/FoEXxwJ2
                                         /entities/NginxController_CZ8QUVgX/logs/access.log
nginx.log.error:                       /home/vagrant/brooklyn-managed-processes/apps/FoEXxwJ2
                                         /entities/NginxController_CZ8QUVgX/logs/error.log
nginx.pid.file:                        /home/vagrant/brooklyn-managed-processes/apps/FoEXxwJ2
                                         /entities/NginxController_CZ8QUVgX/pid.txt
nginx.url.answers.nicely:               true
proxy.domainName:                      8000
proxy.http.port:                       8443
proxy.https.port:                      http
proxy.pool.targets:                    {"TomcatServerImpl{id=QK6Qjmrw}":"192.168.52.103:8080"}
run.dir:                               /home/vagrant/brooklyn-managed-processes/apps/FoEXxwJ2
                                         /entities/NginxController_CZ8QUVgX
service.isUp:                           true
service.notUp.diagnostics:              {}
service.notUp.indicators:               {}
service.problems:                      {}
service.process.isRunning:              true
service.state:                          RUNNING
service.state.expected:                 running @ 1449314377781 / Sat Dec 05 11:19:37 GMT 2015
softwareprocess.pid.file:
softwareservice.provisioningLocation:   {"type":"org.apache.brooklyn.api.location.Location","i
```

```

webapp.url:      d":"zhYBc6xt"}
                  http://192.168.52.102:8000/

```

Details for an individual **sensor** can be shown by providing the **Sensor** Name as a parameter to the `sensor` command:

```

$ br app WebCluster entity CZ8QUVgX sensor service.state.expected
running @ 1449314377781 / Sat Dec 05 11:19:37 GMT 2015

```

Effectors

The effectors for an application or **entity** can be listed with the `effector` command:

```

$ br app WebCluster effector
Name      Description                                     Parameters
restart    Restart the process/service represented by an entity
start      Start the process/service represented by an entity  locations
stop       Stop the process/service represented by an entity

```

```

$ br app WebCluster entity NginxController:CZ8Q effector
Name      Description                                     Parameters
deploy     Deploys an archive ...
getCurrentConfiguration Gets the current ...
populateServiceNotUpDiagnostics Populates the attribute ...
reload     Forces reload of ...
restart     Restart the process/service ... restartChildren,restartMachine
start      Start the process/service ... locations
stop       Stop the process/service ... stopProcessMode,stopMachineMode
update     Updates the entities ...

```

Details of an individual **effector** can be viewed by using the name as a parameter for the `effector` command:

```

$ br app WebCluster entity NginxController:CZ8Q effector update
Name:      update
Description: Updates the entities configuration, and then forces reload of that configuration
Parameters:

```

An **effector** can be invoked by using the `invoke` command with an **effector**-scope:

```

$ br app WebCluster entity NginxController:CZ8Q effector update invoke

```

Parameters can also be passed to the **effector**:

```

$ br app WebCluster entity NginxController:CZ8Q effector restart invoke -P restartChildren=true

```

If a parameter value is complex or spans multiple lines, it may be provided in a file and used like this:

```

$ br app WebCluster effector start invoke -P locations=@data.txt

```

Shortcut commands are available for the 3 standard effectors of `start`, `restart` and `stop`. These commands can be used directly with an app-scope or **entity**-scope:

```

$ br app WebCluster entity NginxController:CZ8Q restart
$ br app WebCluster stop

```

Policies

The policies associated with an application or **entity** can be listed with the `policy` command:

```
$ br app WebCluster entity NginxController:CZ8Q policy
Id          Name          State
VcZ0cfe0   Controller targets tracker  RUNNING
```

Details of an individual **policy** may be viewed by using the PolicyID as a parameter to the `policy` command:

```
$ br app WebCluster entity NginxController:CZ8Q policy VcZ0cfe0
Name          Value          Description
group          DynamicWebAppClusterImpl{id=Tpbkak4D}  group
notifyOnDuplicates  false          Whether to notify listeners when
a sensor is published with the
same value as last time
sensorsToTrack  [Sensor: host.subnet.hostname
(java.lang.String), Sensor: http.port
(java.lang.Integer)]  Sensors of members to be monitored
(implicitly adds service-up
to this list, but that
behaviour may be deleted in a
subsequent release!)
```

Activities

The activities for an application or **entity** may be listed with the `activity` command:

```
$ br app WebCluster activity
Id          Task          Submitted          Status          Streams
Wb6GV5rt    start          Sat Dec 19 11:08:01 GMT 2015  Completed
q2MbyyTo    invoking start[locations] on 2 nodes  Sat Dec 19 11:08:01 GMT 2015  Completed
```

```
$ br app WebCluster entity NginxController:CZ8Q activity
Id          Task          Submitted          Status          Streams
GVh0pyKG    start          Sun Dec 20 19:18:06 GMT 2015  Completed
WJm908rA    provisioning (FixedListMachineProvisi...  Sun Dec 20 19:18:06 GMT 2015  Completed
L0cKFBrW    pre-start      Sun Dec 20 19:18:06 GMT 2015  Completed
D0Ab2esP    ssh: initializing on-box base dir ./b...  Sun Dec 20 19:18:06 GMT 2015  Completed  env,stderr,std
in,stdout
tumLA04     start (processes)  Sun Dec 20 19:18:06 GMT 2015  Completed
YbF2czKM    copy-pre-install-resources  Sun Dec 20 19:18:06 GMT 2015  Completed
o3YdqxsQ    pre-install     Sun Dec 20 19:18:06 GMT 2015  Completed
TtGw4qMZ    pre-install-command  Sun Dec 20 19:18:06 GMT 2015  Completed
duPv0SDB    setup           Sun Dec 20 19:18:06 GMT 2015  Completed
WlTkbhgw    copy-install-resources  Sun Dec 20 19:18:06 GMT 2015  Completed
ZQtrImn1    install         Sun Dec 20 19:18:06 GMT 2015  Completed
hzi49YD6    ssh: setting up sudo  Sun Dec 20 19:18:06 GMT 2015  Completed  env,stderr,std
in,stdout
eEUHcpfi    ssh: Getting machine details for: Ssh...  Sun Dec 20 19:18:07 GMT 2015  Completed  env,stderr,std
in,stdout
juTe2qLG    ssh: installing NginxControllerImpl{i...  Sun Dec 20 19:18:08 GMT 2015  Completed  env,stderr,std
in,stdout
hXqwEZJ1    post-install-command  Sun Dec 20 19:18:08 GMT 2015  Completed
vZliYwBI    customize       Sun Dec 20 19:18:08 GMT 2015  Completed
04Wwb0bP    ssh: customizing NginxControllerImpl{...  Sun Dec 20 19:18:08 GMT 2015  Completed  env,stderr,std
in,stdout
sDwMSkE2    copy-runtime-resources  Sun Dec 20 19:18:08 GMT 2015  Completed
yDYkdkS8    ssh: create run directory  Sun Dec 20 19:18:08 GMT 2015  Completed  env,stderr,std
in,stdout
```

W7dI8r1c	pre-launch-command	Sun Dec 20 19:18:08 GMT 2015	Completed	
0eZKwM5z	launch	Sun Dec 20 19:18:08 GMT 2015	Completed	
y50Gne5E	scheduled:nginx.url.answers.nicely @ ...	Sun Dec 20 19:18:08 GMT 2015	Scheduler,	
ARTnin6E	scheduled:service.process.isRunning @...	Sun Dec 20 19:18:08 GMT 2015	Scheduler,	
tvZoNUTN	ssh: launching NginxControllerImpl{id...	Sun Dec 20 19:18:08 GMT 2015	Completed	env, stderr, std
in, stdout				
YASrjA4w	post-launch-command	Sun Dec 20 19:18:09 GMT 2015	Completed	
jgLYv8pE	post-launch	Sun Dec 20 19:18:09 GMT 2015	Completed	
UN90cwLS	post-start	Sun Dec 20 19:18:09 GMT 2015	Completed	
nmiv97He	reload	Sun Dec 20 19:18:09 GMT 2015	Completed	
FJfPbNtp	ssh: restarting NginxControllerImpl{i...	Sun Dec 20 19:18:10 GMT 2015	Completed	env, stderr, std
in, stdout				
Xm1tjvKf	update	Sun Dec 20 19:18:40 GMT 2015	Completed	
Row67vfa	reload	Sun Dec 20 19:18:40 GMT 2015	Completed	
r8QZXlxJ	ssh: restarting NginxControllerImpl{i...	Sun Dec 20 19:18:40 GMT 2015	Completed	env, stderr, std
in, stdout				

The detail for an individual activity can be viewed by providing the ActivityID as a parameter to the `activity` command (an app-scope or `entity`-scope is not needed for viewing the details of an activity):

```
$ br activity tvZoNUTN
Id: tvZoNUTN
DisplayName: ssh: launching NginxControllerImpl{id=0xPUBk1p}
Description:
EntityId: 0xPUBk1p
EntityDisplayName: NginxController:0xPU
Submitted: Sun Dec 20 19:18:08 GMT 2015
Started: Sun Dec 20 19:18:08 GMT 2015
Ended: Sun Dec 20 19:18:09 GMT 2015
CurrentStatus: Completed
IsError: false
IsCancelled: false
SubmittedByTask: 0eZKwM5z
Streams: stdin: 1133, stdout: 162, stderr: 0, env 0
DetailedStatus: "Completed after 1.05s

Result: 0"
```

The activity command output shows whether any streams were associated with it. The streams and environment for an activity can be viewed with the commands `stdin`, `stdout`, `stderr` and `env`:

```
$ br activity tvZoNUTN stdin
export RUN_DIR="/home/vagrant/brooklyn-managed-processes/apps/V5GQCpIT/entities/NginxController_0xPUBk1p"
mkdir -p $RUN_DIR
cd $RUN_DIR
cd /home/vagrant/brooklyn-managed-processes/apps/V5GQCpIT/entities/NginxController_0xPUBk1p
{ which "/sbin/nginx" || { EXIT_CODE=$? && ( echo "The required executable \"./sbin/nginx\" does not exist" |
tee /dev/stderr ) && exit $EXIT_CODE ; } ; }
nohup ./sbin/nginx -p /home/vagrant/brooklyn-managed-processes/apps/V5GQCpIT/entities/NginxController_0xPUBk1p/
-c conf/server.conf > /home/vagrant/brooklyn-managed-processes/apps/V5GQCpIT/entities/NginxController_0xPUBk1p
/console 2>&1 &
for i in {1..10}
do
test -f /home/vagrant/brooklyn-managed-processes/apps/V5GQCpIT/entities/NginxController_0xPUBk1p/logs/nginx
.pid && ps -p `cat /home/vagrant/brooklyn-managed-processes/apps/V5GQCpIT/entities/NginxController_0xPUBk1p/log
s/nginx.pid` && exit
sleep 1
done
echo "No explicit error launching nginx but couldn't find process by pid; continuing but may subsequently fail"
cat /home/vagrant/brooklyn-managed-processes/apps/V5GQCpIT/entities/NginxController_0xPUBk1p/console | tee /dev
/stderr
```

```
$ br activity tvZoNUTN stdout
./sbin/nginx
```

```

PID TTY          TIME CMD
6178 ?           00:00:00 nginx
Executed /tmp/brooklyn-20151220-191808796-CaiI-launching_NginxControllerImpl_.sh, result 0

```

The child activities of an activity may be listed by providing an activity-scope for the `activity` command:

```

$ br activity 0eZKwM5z
Id:                0eZKwM5z
DisplayName:       launch
Description:
EntityId:          0xPUBk1p
EntityDisplayName: NginxController:0xPU
Submitted:         Sun Dec 20 19:18:08 GMT 2015
Started:           Sun Dec 20 19:18:08 GMT 2015
Ended:             Sun Dec 20 19:18:09 GMT 2015
CurrentStatus:     Completed
IsError:           false
IsCancelled:       false
SubmittedByTask:   tumLAdo4
Streams:
DetailedStatus:    "Completed after 1.06s"

No return value (null)"

$ br activity 0eZKwM5z activity
Id      Task                                Submitted              Status      Streams
tvZoNUTN ssh: launching NginxControllerImpl{id... Sun Dec 20 19:18:08 GMT 2015 Completed env,stderr,stdin,stdout

```

or by using the `-c` (or `--children`) flag with the `activity` command:

```

$ br activity -c 0eZKwM5z
Id      Task                                Submitted              Status      Streams
tvZoNUTN ssh: launching NginxControllerImpl{id... Sun Dec 20 19:18:08 GMT 2015 Completed env,stderr,stdin,stdout

```

YAML Blueprint

This is the [YAML blueprint](#) used for this document, based on the [web cluster](#) examples.

```

name: WebCluster

location:
  byon:
    user: vagrant
    password: vagrant
    hosts:
      - 192.168.52.101
      - 192.168.52.102
      - 192.168.52.103
      - 192.168.52.104
      - 192.168.52.105

services:
- type: org.apache.brooklyn.entity.webapp.ControlledDynamicWebAppCluster
  name: WebApp
  brooklyn.config:
    wars.root: http://search.maven.org/remotecontent?filepath=org/apache/brooklyn/example/brooklyn-example-hello-world-sql-webapp/0.8.0-incubating/brooklyn-example-hello-world-sql-webapp-0.8.0-incubating.war
    java.sysprops:
      brooklyn.example.db.url: >

```

```
    $brooklyn:formatString("jdbc:%s%s?user=%s&password=%s",
    component("db").attributeWhenReady("datastore.url"),
    "visitors", "brooklyn", $brooklyn:external("brooklyn-demo-sample", "hidden-brooklyn-password"))
brooklyn.policies:
- type: org.apache.brooklyn.policy.autoscaling.AutoScalerPolicy
  brooklyn.config:
    metric: webapp.reqs.perSec.windowed.perNode
    metricLowerBound: 2
    metricUpperBound: 10
    minPoolSize: 1
    maxPoolSize: 2
    resizeUpStabilizationDelay: 1m
    resizeDownStabilizationDelay: 5m

- type: org.apache.brooklyn.entity.database.mysql.MySqlNode
  id: db
  name: WebDB
  brooklyn.config:
    creation.script.password: $brooklyn:external("brooklyn-demo-sample", "hidden-brooklyn-password")
    creationScriptUrl: https://bit.ly/brooklyn-visitors-creation-script
```

GUI Guide

- [Launching](#)
- [Deploying Blueprints](#)
- [Monitoring and Managing Applications](#)
- [Using Policies](#)

Launching

This guide will walk you through connecting to the Brooklyn Server Graphical User Interface and performing various tasks.

For an explanation of common Brooklyn Concepts see the [Brooklyn Concepts Quickstart](#) or see the full guide in the [Brooklyn Concepts](#) chapter of the [User Guide](#).

This guide assumes that you are using Linux or Mac OS X and that Brooklyn Server will be running on your local system.

Launch Apache Brooklyn

If you haven't already done so, you will need to start Brooklyn Server using the commands shown below.

It is not necessary at this time, but depending on what you are going to do, you may wish to set up some other configuration options first,

- [Security](#)
- [Persistence](#)

Now start Brooklyn with the following command:

```
$ cd apache-brooklyn-1.0.0-SNAPSHOT
$ bin/brooklyn launch
```

Please refer to the [Server CLI Reference](#) for details of other possible command line options.

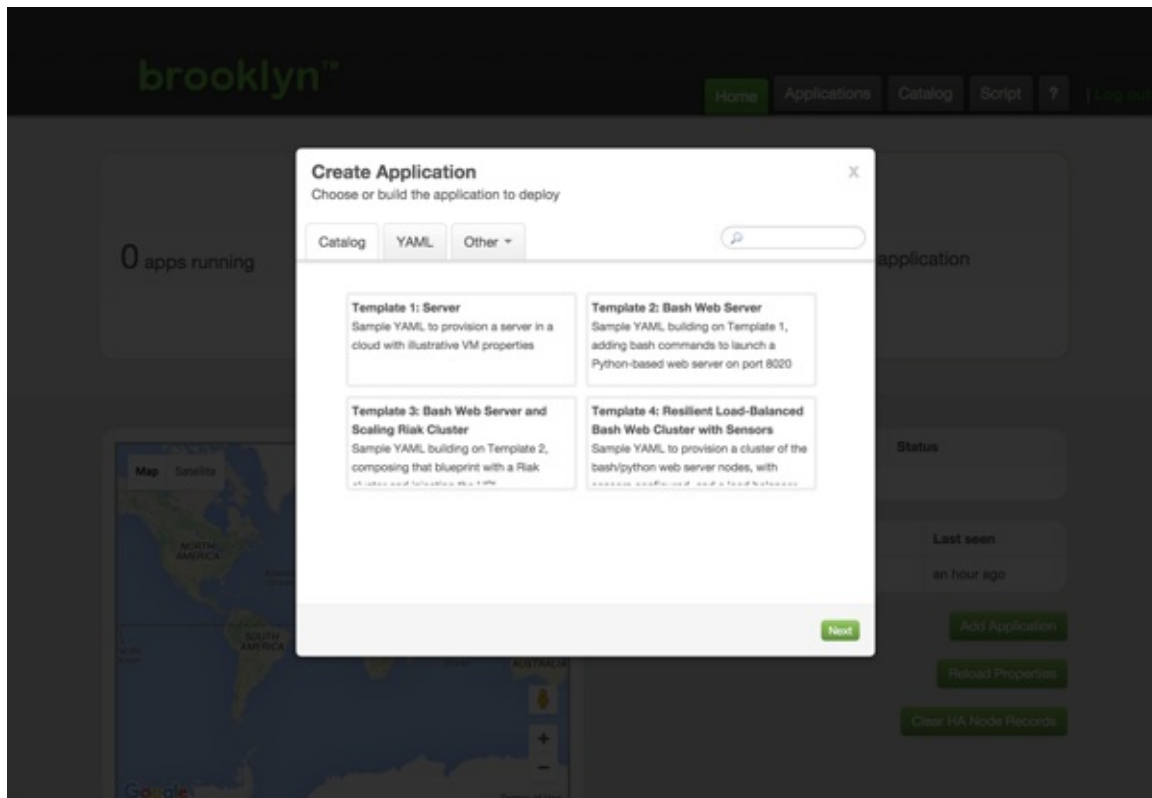
Brooklyn will output the address of the management interface:

```
INFO Starting Brooklyn web-console with no security options (defaulting to no
authentication), on bind address
INFO Started Brooklyn console at http://127.0.0.1:8081/, running
classpath://brooklyn.war@
INFO Persistence disabled
INFO High availability disabled
INFO Launched Brooklyn; will now block until shutdown command received via GUI/API
(recommended) or process interrupt.
```

Notice! Before launching Apache Brooklyn, please check the `date` on the local machine. Even several minutes before or after the actual time could cause problems.

Connect with Browser

Next, open the web console on <http://127.0.0.1:8081>. No applications have been deployed yet, so the "Create Application" dialog opens automatically.



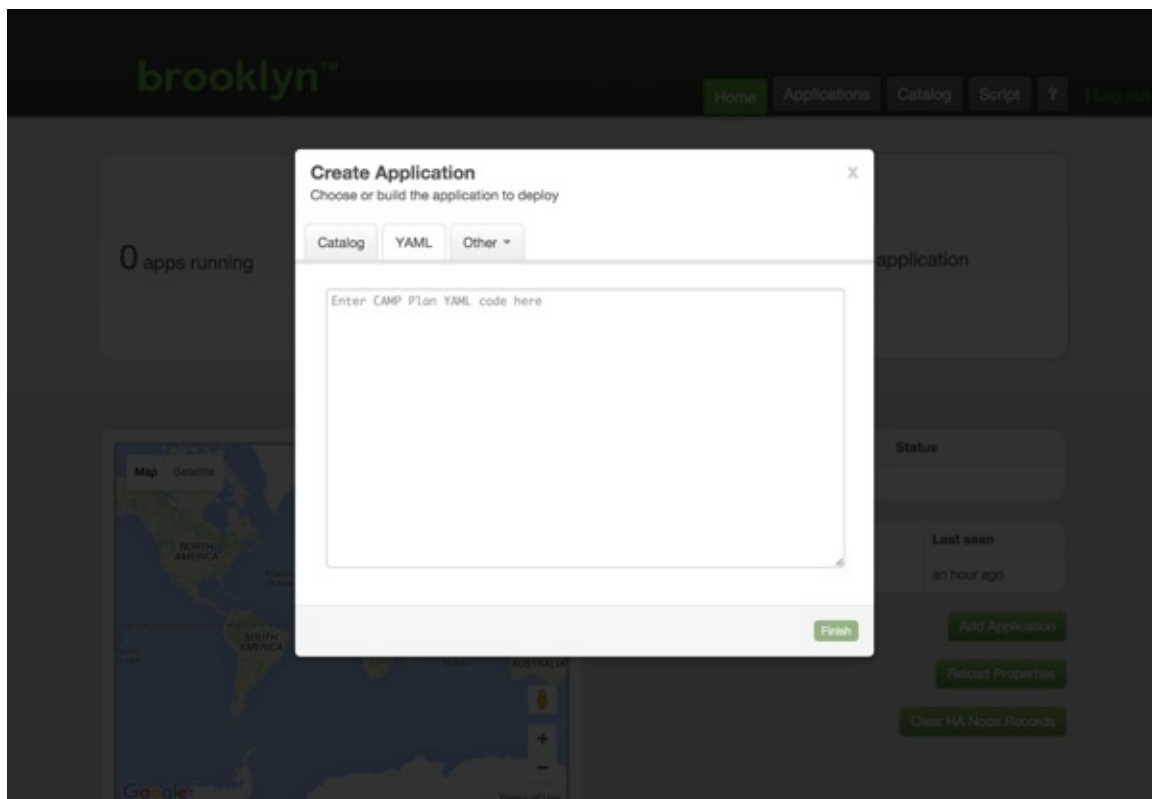
Deploying Blueprints

Launching from a Blueprint

When you first access the web console on <http://127.0.0.1:8081> you will be requested to create your first application.

We'll start by deploying an application via a [YAML blueprint](#) consisting of the following layers.

- MySQL DB
- Dynamic web application cluster
 - Nginx load balancer
 - Tomcat app server cluster



Switch to the [YAML](#) tab and copy the [blueprint](#) below into the large text box.

But *before* you submit it, modify the [YAML](#) to specify the [location](#) where the application will be deployed.

```
name: My Web Cluster

location:
  jclouds:aws-ec2:
    identity: ABCDEFGHIJKLMNOPQRST
    credential: s3cr3tsq1rr3ls3cr3tsq1rr3ls3cr3tsq1rr3l

services:
- type: org.apache.brooklyn.entity.webapp.ControlledDynamicWebAppCluster
  name: My Web
  id: webappcluster
  brooklyn.config:
    wars.root: http://search.maven.org/remotecontent?filepath=org/apache/brooklyn/example/brooklyn-example-hello-world-sql-webapp/0.8.0-incubating/brooklyn-example-hello-world-sql-webapp-0.8.0-incubating.war
```

```

java.sysprops:
  brooklyn.example.db.url: >
    $brooklyn:formatString("jdbc:%s?s?user=%s&password=%s",
      component("db").attributeWhenReady("datastore.url"),
      "visitors", "brooklyn", $brooklyn:external("brooklyn-demo-sample", "hidden-brooklyn-password"))
- type: org.apache.brooklyn.entity.database.mysql.MySqlNode
  name: My DB
  id: db
  brooklyn.config:
    creation.script.password: $brooklyn:external("brooklyn-demo-sample", "hidden-brooklyn-password")
    datastore.creation.script.url: https://bit.ly/brooklyn-visitors-creation-script

```

Replace the `location:` element with values for your chosen target environment, for example to use SoftLayer rather than AWS (updating with your own credentials):

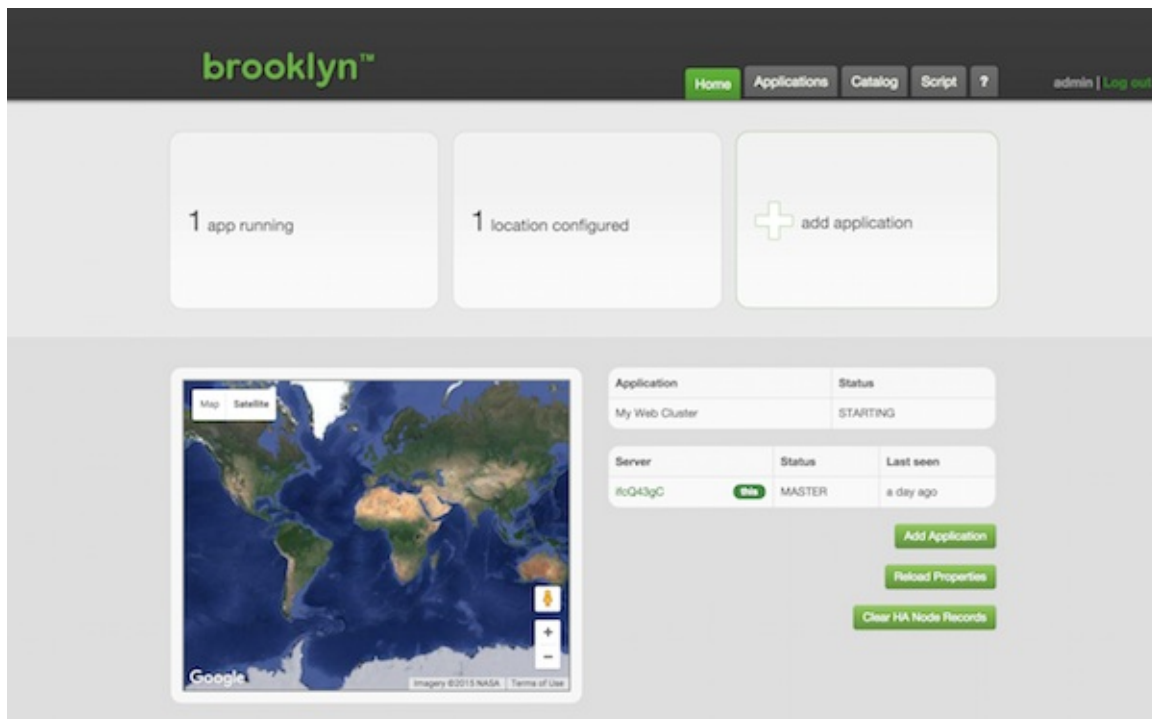
```

location:
  jclouds:softlayer:
    identity: ABCDEFGHIJKLMNOPQRST
    credential: s3cr3tsq1rr3ls3cr3tsq1rr3ls3cr3tsq1rr3l

```

NOTE: See [Locations](#) in the Operations section of the User Guide for instructions on setting up alternate cloud providers, bring-your-own-nodes, or localhost targets, and storing credentials/locations in a file on disk rather than in the [blueprint](#).

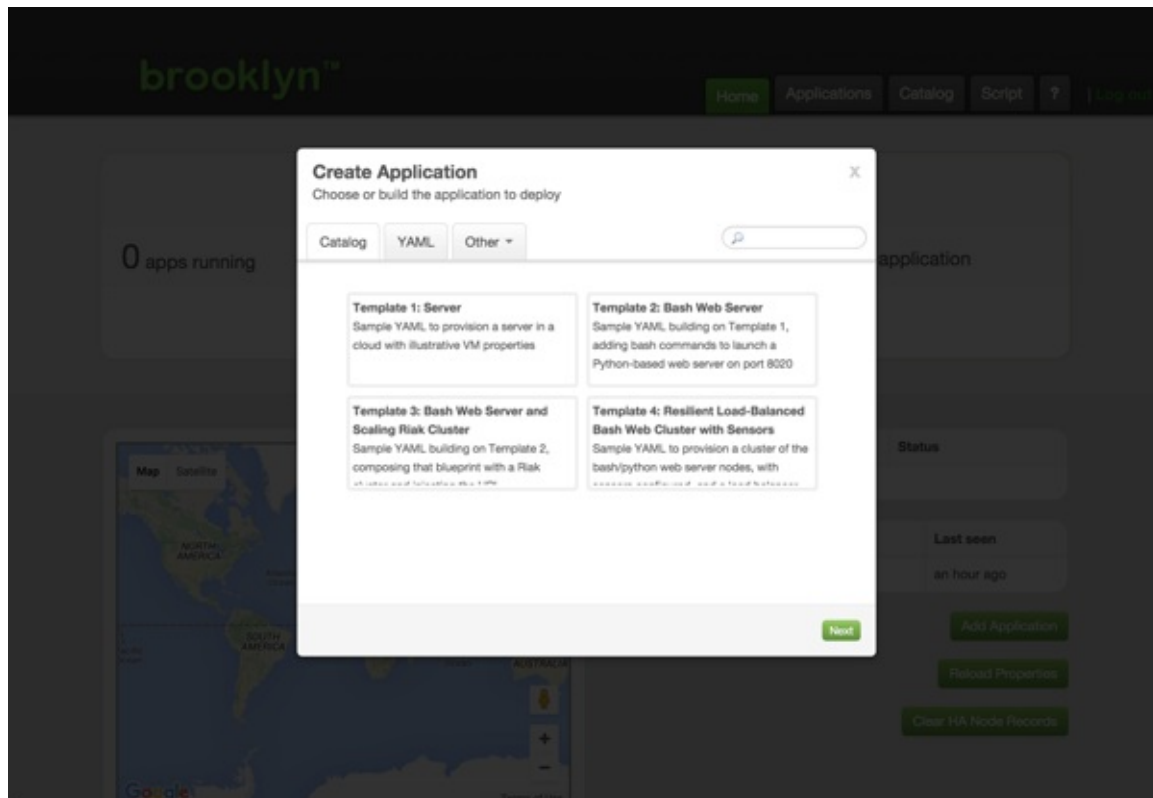
With the modified [YAML](#) in the dialog, click "Finish". The dialog will close and Brooklyn will begin deploying your application. Your application will be shown as "Starting" on the web console's front page.



Depending on your choice of [location](#) it may take some time for the application nodes to start, the next page describes how you can monitor the progress of the application deployment and verify its successful deployment.

Launching from the Catalog

Instead of pasting the [YAML blueprint](#) each time, it can be added to the Brooklyn Catalog where it will be accessible from the Catalog tab of the Create Application dialog.



See **Catalog** in the Operations section of the User Guide for instructions on creating a new Catalog entry from your Blueprint YAML.

Next

So far we have touched on Brooklyn's ability to *deploy* an application **blueprint** to a cloud provider. The next section will show how to **Monitor and Manage Applications**.

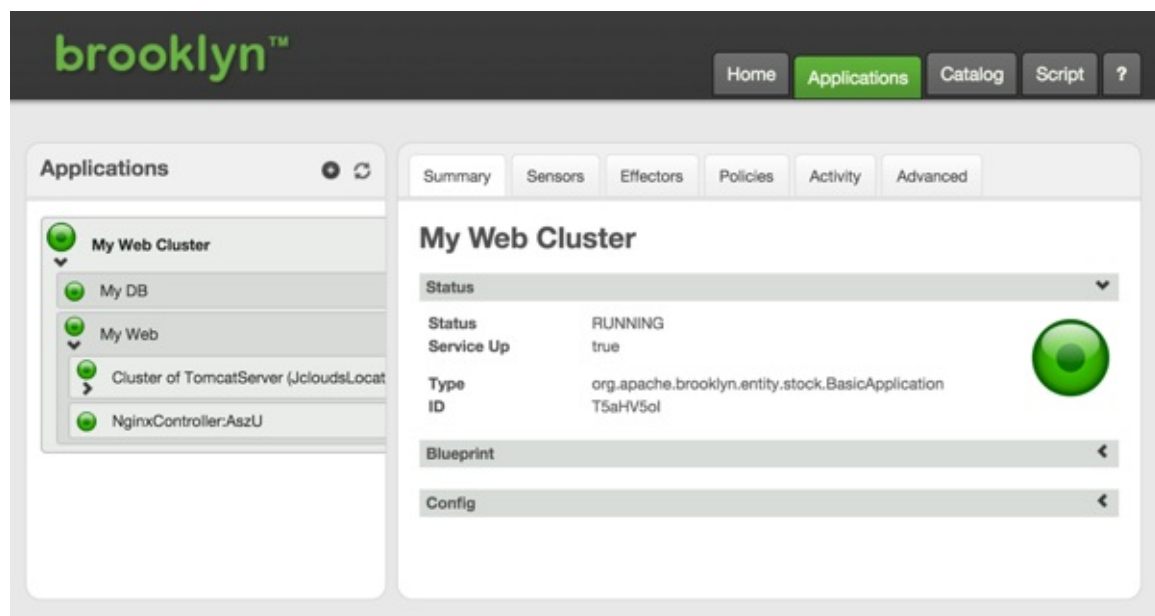
Monitoring and Managing Applications

From the Home page, click on the application name or open the Applications tab.

We can explore the management hierarchy of the application, which will show us the entities it is composed of. Starting from the application use the arrows to expand out the list of entities, or hover over the arrow until a menu popup is displayed so that you can select `Expand All`.

- My Web Cluster (A `BasicApplication`)
 - My DB (A `MySQLNode`)
 - My Web (A `controlledDynamicWebAppCluster`)
 - Cluster of TomcatServer (A `dynamicWebAppCluster`)
 - quarantine (A `QuarantineGroup`)
 - TomcatServer (A `TomcatServer`)
 - NginxController (An `NginxController`)

Clicking on the "My Web Cluster" entity will show the "Summary" tab, giving a very high level of what that component is doing. Click on each of the child components in turn for more detail on that component. Note that the cluster of web servers includes a "quarantine group", to which members of the cluster that fail will be added. These are excluded from the load-balancer's targets.



Activities

The Activity tab allows us to drill down into the tasks each entity is currently executing or has recently completed. It is possible to drill down through all child tasks, and view the commands issued, along with any errors or warnings that occurred.

For example clicking on the NginxController in the left hand tree and opening its Activity tab you can observe the 'start' task is 'In progress'.

Note: You may observe different tasks depending on how far your deployment has progressed).

brooklyn™

Home Applications Catalog Script ?

Applications

Summary Sensors Effectors Policies Activity Advanced

My Web Cluster

- My DB
- My Web
 - Cluster of TomcatServer (JcloudsLocat
 - quarantine
 - TomcatServer:xG8p
 - NginxController:AszU

Task	Submitted	Status
start	Today at 4:33 PM	In progress

Showing 1 - 1 of 1 (25 1 / page)

Clicking on the 'start' task you can discover more details on the actions being carried out by that task (a task may consist of additional subtasks).

brooklyn™

Home Applications Catalog Script ?

Applications

Summary Sensors Effectors Policies Activity Advanced

< NginxController:AszU: start

Invoking effector start on NginxController:AszU with parameters {locations=[JcloudsLocation[aws-ec2:AKIAITAKBYIH2SXH4X3A/aws-ec2@bnap8aCB]]}

In progress (blocked on NginxController:AszU start (processes))

Summary

Name: start
ID: ioU02POI
Submitted: 19 Oct 2015 16:33:37.737 21 minutes ago
Started: 19 Oct 2015 16:33:37.738 21 minutes ago
Submitted by: My Web start (parallel)

Children Tasks

Task	Submitted	Status
provisioning (aws-ec2)	Today at 4:33 PM	Completed
pre-start	Today at 4:44 PM	Completed
start (processes)	Today at 4:49 PM	In progress
post-start	-	Not submitted

Showing 1 - 4 of 4 (25 1 / page)

Background Tasks

Tags

Detailed Status

JSON

Continuing to drill down into the 'In progress' tasks you will eventually reach the currently active task where you can investigate the ssh command executed on the target node including the current stdin, stdout and stderr output.

The screenshot displays the Brooklyn UI interface. On the left, the 'Applications' sidebar shows a hierarchical tree view. The 'My Web Cluster' is expanded, revealing sub-entities like 'My DB', 'My Web', 'Cluster of TomcatServer (JcloudsLocat', and 'NginxController:AszU'. The main content area is titled 'Applications' and features a tabbed interface with 'Summary', 'Sensors', 'Effectors', 'Policies', 'Activity', and 'Advanced'. The 'Summary' tab is active, showing details for the task 'NginxController:AszU: ssh: customizing NginxControllerImpl[id=AszUfOjs]'. The task status is 'In progress - Opening ssh connection'. The summary includes fields for Name, ID, Submitted time, Started time, and Submitted by. Below this, the 'Available Streams' section lists 'env', 'stderr', 'stdin', and 'stdout' with download links and sizes. Further down, there are expandable sections for 'Background Tasks', 'Tags', 'Detailed Status', and 'JSON'.

Sensors

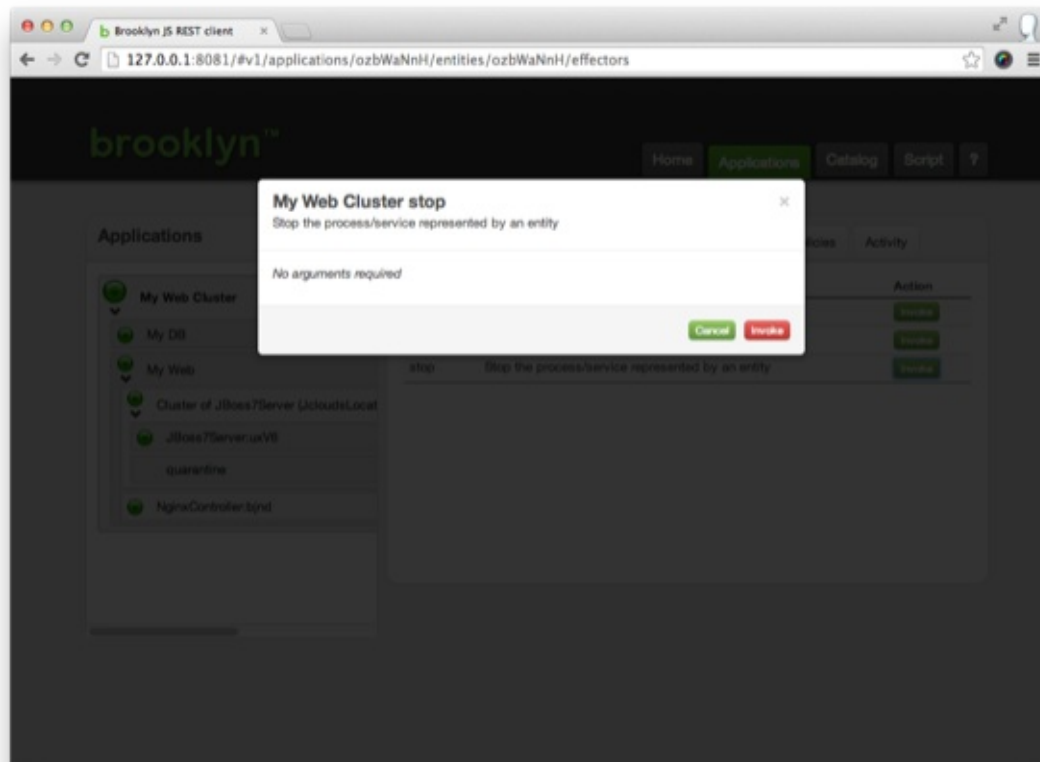
Now click on the "Sensors" tab: these data feeds drive the real-time picture of the application. As you navigate in the tree at the left, you can see more targeted statistics coming in in real-time.

Explore the sensors and the tree to find the URL where the *NginxController* for the webapp we just deployed is running. This can be found in '**My Web Cluster -> My Web -> NginxController -> main.uri**'.

Quickly return to the '**Brooklyn JS REST client**' web browser tab showing the "Sensors" and observe the '**My Web Cluster -> My Web -> Cluster of TomcatServer -> webapp reqs.perSec.last**' sensor value increase.

Stopping the Application

To stop an application, select the application in the tree view (the top/root entity), click on the Effectors tab, and invoke the "Stop" effector. This will cleanly shutdown all components in the application and return any cloud machines that were being used.



Next

Brooklyn's real power is in using **Policies** to automatically *manage* applications.

Using Policies

Exploring and Testing Policies

To see an example of [policy](#) based management, please deploy the following [blueprint](#) (changing the [location](#) details as for the example shown earlier):

```
name: My Web Cluster

location: localhost

services:

- type: org.apache.brooklyn.entity.webapp.ControlledDynamicWebAppCluster
  name: My Web
  brooklyn.config:
    wars.root: http://search.maven.org/remotecontent?filepath=org/apache/brooklyn/example/brooklyn-example-hello-world-sql-webapp/0.8.0-incubating/brooklyn-example-hello-world-sql-webapp-0.8.0-incubating.war
    java.sysprops:
      brooklyn.example.db.url: >
        $brooklyn:formatString("jdbc:%s%s?user=%s&password=%s",
          component("db").attributeWhenReady("datastore.url"),
          "visitors", "brooklyn", $brooklyn:external("brooklyn-demo-sample", "hidden-brooklyn-password"))
  brooklyn.policies:
- type: org.apache.brooklyn.policy.autoscaling.AutoScalerPolicy
  brooklyn.config:
    metric: webapp.reqs.perSec.windowed.perNode
    metricLowerBound: 0.1
    metricUpperBound: 10
    minPoolSize: 1
    maxPoolSize: 4
    resizeUpStabilizationDelay: 10s
    resizeDownStabilizationDelay: 1m

- type: org.apache.brooklyn.entity.database.mysql.MySqlNode
  id: db
  name: My DB
  brooklyn.config:
    creation.script.password: $brooklyn:external("brooklyn-demo-sample", "hidden-brooklyn-password")
    datastore.creation.script.url: https://bit.ly/brooklyn-visitors-creation-script
```

The app server cluster has an `AutoScalerPolicy`, and the loadbalancer has a `targets` [policy](#).

Use the Applications tab in the web console to drill down into the Policies section of the `ControlledDynamicWebAppCluster`. You will see that the `AutoScalerPolicy` is running.

This [policy](#) automatically scales the cluster up or down to be the right size for the cluster's current load. One server is the minimum size allowed by the [policy](#).

The loadbalancer's `targets` [policy](#) ensures that the loadbalancer is updated as the cluster size changes.

Sitting idle, this cluster will only contain one server, but you can use a tool like [jmeter](#) pointed at the nginx endpoint to create load on the cluster. Download a jmeter test plan [here](#).

As load is added, Apache Brooklyn requests a new cloud machine, creates a new app server, and adds it to the cluster. As load is removed, servers are removed from the cluster, and the infrastructure is handed back to the cloud.

Under the Covers

The `AutoScalerPolicy` here is configured to respond to the `sensor` reporting requests per second per node, invoking the default `resize` `effector`. By clicking on the `policy`, you can configure it to respond to a much lower threshold or set long stabilization delays (the period before it scales out or back).

An even simpler test is to manually suspend the `policy`, by clicking "Suspend" in the policies list. You can then switch to the "Effectors" tab and manually trigger a `resize`. On resize, new nodes are created and configured, and in this case a `policy` on the nginx node reconfigures nginx whenever the set of active targets changes.

Next

This guide has given a quick overview of using the Apache Brooklyn GUI to deploy, monitor and manage applications. The GUI also allows you to perform various Advanced management tasks and to explore and use the REST API (from the Script tab). Please take some time now to become more familiar with the GUI.

Then continue to read through the [Operations Guide](#).

REST API

Apache Brooklyn exposes a powerful REST API, allowing it to be scripted from bash or integrated with other systems.

For many commands, the REST call follows the same structure as the web console URL scheme, but with the `#` at the start of the path removed; for instance the catalog item `cluster` in the web console is displayed at:

```
http://localhost:8081/#v1/catalog/entities/cluster:1.0.0-SNAPSHOT
```

And in the REST API it is accessed at:

```
http://localhost:8081/v1/catalog/entities/cluster:1.0.0-SNAPSHOT
```

A full reference for the REST API is automatically generated by the server at runtime. It can be found in the Brooklyn web console, under the Script tab.

Here we include some of the most common REST examples and other advice for working with the REST API.

Tooling Tips

For command-line access, we recommend `curl`, with tips below.

For navigating in a browser we recommend getting a plugin for working with REST; these are available for most browsers and make it easier to authenticate, set headers, and see JSON responses.

For manipulating JSON responses on the command-line, the library `jq` from [stedolan's github](#) is very useful, and available in most package repositories, including `port` and `brew` on Mac.

Common REST Examples

Here are some useful snippets:

- List applications

```
curl http://localhost:8081/v1/applications
```

- Deploy an application from `__FILE__`

```
curl http://localhost:8081/v1/applications --data-binary @__FILE__
```

- Get details of a task with ID `__ID__` (where the `id` is returned by the above, optionally piped to `jq .id`)

```
curl http://localhost:8081/v1/activities/__ID__
```

- Get the value of `sensor` `service.state` on `entity` `e1` in application `app1` (note you can use either the `entity`'s ID or its name)

```
curl http://localhost:8081/v1/applications/app1/entities/e1/sensors/service.state
```

- Get all `sensor` values (using the pseudo-`sensor` `current-state`)

```
curl http://localhost:8081/v1/applications/app1/entities/e1/sensors/service.state
```

- Invoke an **effector** `eff` on `e1`, with argument `arg1` equal to `hi` (note if no arguments, you must specify `-d ""`; for multiple args, just use multiple `-d` entries, or a JSON file with `--data-binary @...`)

```
curl http://localhost:8081/v1/applications/app1/entities/e1/effectors/eff -d arg1=hi
```

- Add an item to the catalog from `__FILE__`

```
curl http://localhost:8081/v1/catalog --data-binary @__FILE__
```

Curl Cheat Sheet

- if authentication is required, use `--user username:password`
- to see detailed output, including headers, code, and errors, use `-v`
- where the request is not a simple HTTP GET, use `-X POST` or `-X DELETE`
- to pass key-value data to a post, use `-d key=value`
- where you are posting from a file `__FILE__`, use `--data-binary @__FILE__` (implies a POST) or `-T __FILE__ -X POST`
- to add a header, use `-H "key: value"`, for example `-H "Brooklyn-Allow-Non-Master-Access: true"`
- to specify that a specific content-type is being uploaded, use `-H "Content-Type: application/json"` (or `application/yaml`)
- to specify the content-type required for the result, use `-H "Accept: application/json"` (or `application/yaml`, or for **sensor** values, `text/plain`)

Configuring Brooklyn

Apache Brooklyn contains a number of configuration options managed across several files. Historically Brooklyn has been configured through a `brooklyn.properties` file, this changed to a `brooklyn.cfg` file when the Karaf release became the default in Brooklyn 0.12.0.

The configurations for [persistence](#) and [high availability](#) are described elsewhere in this manual.

Configuration of Apache Brooklyn when running under Karaf is largely done through standard Karaf mechanisms. The Karaf "Configuration Admin" subsystem is used to manage configuration values loaded at first boot from the `.cfg` files in the `etc` directory of the distribution. In the Karaf command line these can then be viewed and manipulated by the `config:` commands, see the [Karaf documentation](#) for full details.

Configuring Brooklyn Properties

To configure the Brooklyn runtime create an `etc/brooklyn.cfg` file. If you have previously used `brooklyn.properties` it follows the same file format. Values can be viewed and managed dynamically via the OSGI configuration admin commands in Karaf, e.g. `config:property-set`. The global `~/.brooklyn/brooklyn.properties` is still supported and has higher priority for duplicate keys, but its values can't be manipulated with the Karaf commands, so its use is discouraged.

You can use the standard `~/.brooklyn/brooklyn.properties` file to configure Brooklyn. Alternatively create `etc/brooklyn.cfg` inside the distribution folder (same file format). The keys in the former override those in the latter.

Web console related configuration is done through the corresponding Karaf mechanisms:

- The port is set in `etc/org.ops4j.pax.web.cfg`, key `org.osgi.service.http.port`.
- For authentication the JAAS realm "webconsole" is used; by default it will use any SecurityProvider implementations configured in Brooklyn falling back to auto generating the password. To configure a custom JAAS realm see the `jetty.xml` file in `brooklyn-server/karaf/jetty-config/src/main/resources` and override it by creating a custom one in `etc` folder. Point the "webconsole" login service to the JAAS realm you would like to use.
 - For other Jetty related configuration consult the Karaf and pax-web docs.

Memory Usage

The amount of memory required by Apache Brooklyn process depends on the usage - for example the number of entities/VMs under management.

For a standard Brooklyn deployment, the defaults are to start with 256m, and to grow to 2g of memory. These numbers can be overridden by setting the `JAVA_MAX_MEM` and `JAVA_MAX_PERM_MEM` in the `bin/setenv` script:

```
export JAVA_MAX_MEM="2G"
```

Apache Brooklyn stores a task history in-memory using [soft references](#). This means that, once the task history is large, Brooklyn will continually use the maximum allocated memory. It will only expunge tasks from memory when this space is required for other objects within the Brooklyn process.

Authentication and Security

There are two areas of authentication used in Apache Brooklyn, these are as follows:

- Karaf authentication

Apache Brooklyn uses [Apache Karaf](#) as a core platform, this has user level security and groups which can be configured as detailed [here](#).

- Apache Brooklyn authentication

Users and passwords for Brooklyn can be configured in the `brooklyn.cfg` as detailed [here](#).

HTTPS Configuration

See [HTTPS Configuration](#) for general information on configuring HTTPS.

Catalog in Karaf

With the traditional launcher, Brooklyn loads the initial contents of the catalog from a `default.catalog.bom` file as described in the section on [installation](#). Brooklyn finds Java implementations to provide for certain things in blueprints (entities, enrichers etc.) by scanning the classpath.

In the OSGI world this approach is not used, as each bundle only has visibility of its own and its imported Java packages. Instead, in Karaf, each bundle can declare its own `catalog.bom` file, in the root of the bundle, with the catalog declarations for any entities etc. that the bundle contains.

For example, the `catalog.bom` file for Brooklyn's Webapp bundle looks like (abbreviated):

```
brooklyn.catalog:
  version: ...
  items:
    - id: org.apache.brooklyn.entity.webapp.nodejs.NodeJsWebAppService
      itemType: entity
      item:
        type: org.apache.brooklyn.entity.webapp.nodejs.NodeJsWebAppService
        name: Node.JS Application
    ...
    - id: resilient-bash-web-cluster-template
      itemType: template
      name: "Template: Resilient Load-Balanced Bash Web Cluster with Sensors"
      description: |
        Sample YAML to provision a cluster of the bash/python web server nodes,
        with sensors configured, and a load balancer pointing at them,
        and resilience policies for node replacement and scaling
      item:
        name: Resilient Load-Balanced Bash Web Cluster (Brooklyn Example)
```

In the above [YAML](#) the first item declares that the bundle provides an [entity](#) whose type is `org.apache.brooklyn.entity.webapp.nodejs.NodeJsWebAppService`, and whose name is 'Node.JS Application'. The second item declares that the bundle provides a template application, with id `resilient-bash-web-cluster-template`, and includes a description for what this is.

Configuring the applications in the Catalog

When running some particular deployment of Brooklyn it may not be desirable for the sample applications to appear in the catalog (for clarity, "application" here in the sense of an item with `itemType: template`). For example, if you have developed some bundle with your own application and added it to Karaf then you might want only your own application to appear in the catalog.

Brooklyn contains a mechanism to allow you to configure what bundles will add their applications to the catalog. The Karaf configuration file `/etc/org.apache.brooklyn.core.catalog.bomscanner.cfg` contains two properties, one `whitelist` and the other `blacklist`, that bundles must satisfy for their applications to be added to the catalog. Each property value is a comma-separated list of regular expressions. The symbolic id of the bundle must match one of the regular expressions on the whitelist, and not match any expression on the blacklist, if its applications are to be added to the bundle. The default values of these properties are to admit all bundles, and forbid none.

Caveats

In the OSGi world specifying class names by string in Brooklyn's configuration will work only for classes living in Brooklyn's core modules. Raise an issue or ping us on IRC if you find a case where this doesn't work for you. For custom `SecurityProvider` implementations refer to the documentation of `BrooklynLoginModule`.

- [Memory Usage](#)
- [Authentication](#)
- [brooklyn.cfg](#)
- [HTTPS Configuration](#)
- [CORS Configuration](#)

brooklyn.cfg

The file `brooklyn.cfg` is read when Apache Brooklyn starts in order to load any server configuration values. It can be found in the Brooklyn configuration folder. You can check [here](#) for the [location](#) of your Brooklyn configuration folder

Quick Setup

The most common properties set in this file are for access control. Without this, Brooklyn's web-console and REST api will require no authentication.

The simplest way to specify users and passwords is shown below (but see the [Authentication](#) section for how to avoid storing passwords in plain text):

```
brooklyn.webconsole.security.users=admin,bob
brooklyn.webconsole.security.user.admin.password=AdminPassw0rd
brooklyn.webconsole.security.user.bob.password=BobPassw0rd
```

In many cases, it is preferable instead to use an external credentials store such as LDAP. Information on configuring these is [below](#).

If coming over a network it is highly recommended additionally to use `https`. This can be configured with:

```
brooklyn.webconsole.security.https.required=true
```

More information, including setting up a certificate, is described [further below](#).

Camp YAML Expressions

Values in `brooklyn.cfg` can use the Camp [YAML](#) syntax. Any value starting `$brooklyn:` is parsed as a Camp [YAML](#) expression.

This allows [externalized configuration](#) to be used from `brooklyn.cfg`. For example:

```
brooklyn.location.jclouds.aws-ec2.identity=$brooklyn:external("vault", "aws-identity")
brooklyn.location.jclouds.aws-ec2.credential=$brooklyn:external("vault", "aws-credential")
```

If for some reason one requires a literal value that really does start with `$brooklyn:` (i.e. for the value to not be parsed), then this can be achieved by using the syntax below. This example returns the property value `$brooklyn:myexample` :

```
example.property=$brooklyn:literal("$brooklyn:myexample")
```

Java

Arbitrary data can be set in the `brooklyn.cfg`. This can be accessed in java using `ManagementContext.getConfig(KEY)`.

Authentication

Security Providers are the mechanism by which different authentication authorities are plugged in to Brooklyn. These can be configured by specifying `brooklyn.webconsole.security.provider` equal to the name of a class implementing `SecurityProvider`. An implementation of this could point to Spring, LDAP, OpenID or another identity management system.

The default implementation, `ExplicitUsersSecurityProvider`, reads from a list of users and passwords which should be specified as configuration parameters e.g. in `brooklyn.cfg`. This configuration could look like:

```
brooklyn.webconsole.security.users=admin
brooklyn.webconsole.security.user.admin.salt=0Hdf
brooklyn.webconsole.security.user.admin.sha256=91e16f94509fa8e3dd21c43d69cadfd7da6e7384051b18f168390fe378bb36f9
```

The `users` line should contain a comma-separated list. The special value `*` is accepted to permit all users.

To generate this, the brooklyn CLI can be used:

```
brooklyn generate-password --user admin

Enter password:
Re-enter password:

Please add the following to your brooklyn.properties:

brooklyn.webconsole.security.users=admin
brooklyn.webconsole.security.user.admin.salt=0Hdf
brooklyn.webconsole.security.user.admin.sha256=91e16f94509fa8e3dd21c43d69cadfd7da6e7384051b18f168390fe378bb36f9
```

Alternatively, in dev/test environments where a lower level of security is required, the syntax

`brooklyn.webconsole.security.user.<username>=<password>` can be used for each `<username>` specified in the `brooklyn.webconsole.security.users` list.

Other security providers available include:

No one

`brooklyn.webconsole.security.provider=org.apache.brooklyn.rest.security.provider.BlackholeSecurityProvider` will block all logins (e.g. if not using the web console)

No security

`brooklyn.webconsole.security.provider=org.apache.brooklyn.rest.security.provider.AnyoneSecurityProvider` will allow logins with no credentials (e.g. in secure dev/test environments)

LDAP

`brooklyn.webconsole.security.provider=org.apache.brooklyn.rest.security.provider.LdapSecurityProvider` will cause Brooklyn to call to an LDAP server to authenticate users; The other things you need to set in `brooklyn.cfg` are:

- `brooklyn.webconsole.security.ldap.url` - ldap connection url
- `brooklyn.webconsole.security.ldap.realm` - ldap dc parameter (domain)
- `brooklyn.webconsole.security.ldap.ou` *optional, by default it set to Users* - ldap ou parameter

brooklyn.cfg example configuration:

```
brooklyn.webconsole.security.provider=org.apache.brooklyn.rest.security.provider.LdapSecurityProvider
```

```
brooklyn.webconsole.security.ldap.url=ldap://localhost:10389/???X-BIND-USER=uid=admin%2cou=system,X-BIND-PASSW
ORD=secret,X-COUNT-LIMIT=1000
brooklyn.webconsole.security.ldap.realm=example.com
```

After you setup the brooklyn connection to your LDAP server, you can authenticate in brooklyn using your cn (e.g. John Smith) and your password. `org.apache.brooklyn.rest.security.provider.LdapSecurityProvider` searches in the LDAP tree in `LDAP://cn=John Smith,ou=Users,dc=example,dc=com`

If you want to customize the ldap path or something else which is particular to your LDAP setup you can extend `LdapSecurityProvider` class or implement from scratch the `SecurityProvider` interface.

Entitlements

In addition to login access, fine-grained permissions -- including seeing entities, creating applications, seeing sensors, and invoking effectors -- can be defined on a per-user *and* per-target (e.g. which [entity/effector](#)) basis using a plug-in **Entitlement Manager**.

This can be set globally with the property:

```
brooklyn.entitlements.global=<class>
```

The default entitlement manager is one which responds to per-user entitlement rules, and understands:

- `root` : full access, including to the Groovy console
- `user` : access to everything but actions that affect the server itself. Such actions include the Groovy console, stopping the server and retrieving management context configuration.
- `readonly` : read-only access to almost all information
- `minimal` : access only to server stats, for use by monitoring systems

These keywords are also understood at the `global` level, so to grant full access to `admin`, read-only access to `support`, limited access to `metrics` and regular access to `user` you can write:

```
brooklyn.entitlements.global=user
brooklyn.entitlements.perUser.admin=root
brooklyn.entitlements.perUser.support=readonly
brooklyn.entitlements.perUser.metrics=minimal
```

Under the covers this invokes the `PerUserEntitlementManager`, with a `default` set (and if not specified `default` defaults to `minimal`); so the above can equivalently be written:

```
brooklyn.entitlements.global=org.apache.brooklyn.core.mgmt.entitlement.PerUserEntitlementManager
brooklyn.entitlements.perUser.default=user
brooklyn.entitlements.perUser.admin=root
brooklyn.entitlements.perUser.support=readonly
brooklyn.entitlements.perUser.metrics=minimal
```

For more information, see [Java: Entitlements](#). or [EntitlementManager](#).

HTTPS Configuration

See [HTTPS Configuration](#) for general information on configuring HTTPS.

HTTPS Configuration

Getting the Certificate

To enable HTTPS web access, you will need a server certificate in a Java keystore. To create a self-signed certificate, for testing and non-production use, you can use the tool `keytool` from your Java distribution. (A self-signed certificate will cause a warning to be displayed by a browser when viewing the page. The various browsers each have ways to import the certificate as a trusted one, for test purposes.)

The following command creates a self-signed certificate and adds it to a keystore, `keystore.jks` :

```
% keytool -genkey -keyalg RSA -alias brooklyn \
  -keystore <path-to-keystore-directory>/keystore.jks -storepass "mypassword" \
  -validity 365 -keysize 2048 -keypass "password"
```

The passwords above should be changed to your own values. Omit those arguments above for the tool to prompt you for the values.

You will then be prompted to enter your name and organization details. This will use (or create, if it does not exist) a keystore with the password `mypassword` - you should use your own secure password, which will be the same password used in your `brooklyn.cfg` (below). You will also need to replace `<path-to-keystore-directory>` with the full path of the folder where you wish to store your keystore. The keystore will contain the newly generated key, with alias `brooklyn` and password `password` .

For production servers, a valid signed certificate from a trusted certifying authority should be used instead. Typically keys from a certifying authority are not provided in Java keystore format. To create a Java keystore from existing certificates (CA certificate, and public and private keys), you must first create a PKCS12 keystore from them, for example with `openssl` ; this can then be converted into a Java keystore with `keytool` . For example, with a CA certificate `ca.pem` , and public and private keys `cert.pem` and `key.pem` , create the PKCS12 store `server.p12` , and then convert it into a keystore `keystore.jks` as follows:

```
% openssl pkcs12 -export -in cert.pem -inkey key.pem \
  -out server.p12 -name "brooklyn" \
  -CAfile ca.pem -caname root -chain -passout pass:"password"

% keytool -importkeystore \
  -deststorepass "password" -destkeypass "password" -destkeystore keystore.jks \
  -srckeystore server.p12 -srcstoretype PKCS12 -srcstorepass "password" \
  -alias "brooklyn"
```

HTTPS Configuration

In `org.ops4j.pax.web.cfg` in the Brooklyn distribution root, un-comment the settings:

```
org.osgi.service.http.port.secure=8443
org.osgi.service.http.secure.enabled=true
org.ops4j.pax.web.ssl.keystore=${karaf.home}/etc/keystores/keystore.jks
org.ops4j.pax.web.ssl.password=password
org.ops4j.pax.web.ssl.keypassword=password
org.ops4j.pax.web.ssl.clientauthwanted=false
org.ops4j.pax.web.ssl.clientauthneeded=false
```

replacing the passwords with appropriate values, and restart the server. Note the keystore [location](#) is relative to the installation root, but a fully qualified path can also be given, if it is desired to use some separate pre-existing store.

CORS Configuration

To enable / configure [cross-origin resource sharing \(CORS\)](#). The following file must be added to [org.apache.brooklyn.rest.filter.cors.cfg](#)

```
# Enables experimental support for Cross Origin Resource Sharing (CORS) filtering in Apache Brooklyn REST API.
cors.enabled=true

# @see CrossOriginResourceSharingFilter#setAllowOrigins(List<String>)
# Coma separated values list of allowed origins. Access-Control-Allow-Origin header will be returned to client
if Origin header in request is matching exactly a value among the list allowed origins.
# If empty or not specified then all origins are allowed. No wildcard allowed origins are supported.
cors.allow.origins=http://host-one.example.com:8080, http://host-two.example.com, https://host-three.example.co
m

# @see CrossOriginResourceSharingFilter#setAllowHeaders(List<String>)
# Coma separated values list of allowed headers for preflight checks.
#cors.allow.headers=

# @see CrossOriginResourceSharingFilter#setAllowCredentials(boolean)
# The value for the Access-Control-Allow-Credentials header. If false, no header is added.
# If true, the header is added with the value 'true'. False by default.
#cors.allow.credentials=false

# @see CrossOriginResourceSharingFilter#setExposeHeaders(List<String>)
# CSV list of non-simple headers to be exposed via Access-Control-Expose-Headers.
#cors.expose.headers=

# @see CrossOriginResourceSharingFilter#setMaxAge(Integer)
# The value for Access-Control-Max-Age. If -1 then No Access-Control-Max-Age header will be send.
#cors.max.age=-1

# @see CrossOriginResourceSharingFilter#setPreflightErrorStatus(Integer)
# Preflight error response status, default is 200.
cors.preflight.error.status=200

# Do not apply CORS if response is going to be with UNAUTHORIZED status.
#cors.block.if.unauthorized=false
```

NOTE: You must [restart Brooklyn](#) for these changes to be applied

Further information on client side [usage](#)

Persistence

By default Brooklyn persists its state to storage so that a server can be restarted without loss or so a high availability standby server can take over.

Brooklyn can persist its state to one of two places: the file system, or to an [object store](#) of your choice.

Configuration

To configure persistence, edit the file `org.apache.brooklyn.osgi.launcher.cfg` in the `etc` directory of your Brooklyn instance. The following options are available:

`persistMode` - This is the mode in which persistence is running, in and is set to `auto` by default. The possible values are:

- `AUTO` - will rebind to any existing state, or start up fresh if no state;
- `DISABLED` - will not read or persist any state;
- `REBIND` - will rebind to the existing state, or fail if no state available;
- `CLEAN` - will start up fresh (removing any existing state)

`persistenceDir` - This is the directory to which Apache Brooklyn reads and writes its persistence data. The default [location](#) depends on your installation method. Checkout [this page](#) for more information.

`persistenceLocation` - This is the [location](#) for an object store to read and write persisted state.

`persistPeriod` - This is an interval period which can be set to reduce the frequency with which persistence is carried out, for example `1s`.

File-based Persistence

Apache Brooklyn starts with file-based persistence by default, saving data in the [persisted state folder](#). For the rest of this document we will refer to this [location](#) as `%persistence-home%`.

If there is already data at `%persistence-home%/data`, then a backup of the directory will be made. This will have a name like `%persistence-home%/backups/%date%-%time%-jvyX7Wis-promotion-igFH`. This means backups of the data directory will be automatically created each time Brooklyn is restarted (or if a standby Brooklyn instances takes over as master).

The state is written to the given path. The file structure under that path is:

- `./catalog/`
- `./enrichers/`
- `./entities/`
- `./feeds/`
- `./locations/`
- `./nodes/`
- `./plane/`
- `./policies/`

In each of those directories, an XML file will be created per item - for example a file per [entity](#) in `./entities/`. This file will capture all of the state - for example, an [entity](#)'s: id; display name; type; config; attributes; tags; relationships to locations, child entities, group membership, policies and enrichers; and dynamically added effectors and sensors.

Object Store Persistence

Apache Brooklyn can persist its state to any Object Store API supported by [Apache jclouds](#) including [S3](#), [Swift](#) and [Azure](#). This gives access to any compatible Object Store product or cloud provider including AWS-S3, SoftLayer, Rackspace, HP and Microsoft Azure. For a complete list of supported providers, see [jclouds](#).

To configure the Object Store, add the credentials to `brooklyn.cfg` such as:

```
brooklyn.location.named.aws-s3-eu-west-1=aws-s3:eu-west-1
brooklyn.location.named.aws-s3-eu-west-1.identity=ABCDEFGHJKLMNOPQRSTU
brooklyn.location.named.aws-s3-eu-west-1.credential=abcdefghijklmnopqrstuvwxyz1234567890ab/c
```

or:

```
brooklyn.location.named.softlayer-swift-ams01=jclouds:openstack-swift:https://ams01.objectstorage.softlayer.net/auth/v1.0
brooklyn.location.named.softlayer-swift-ams01.identity=ABCDEFGHJKLM:myname
brooklyn.location.named.softlayer-swift-ams01.credential=abcdefghijklmnopqrstuvwxyz1234567890abcdefghijklmnopqrstuvwxyz12
brooklyn.location.named.softlayer-swift-ams01.jclouds.keystone.credential-type=tempAuthCredentials
```

Then edit the `persistenceLocation` to point at this object store: `softlayer-swift-ams01`.

Rebinding to State

When Brooklyn starts up pointing at existing state, it will recreate the entities, locations and policies based on that persisted state.

Once all have been created, Brooklyn will "manage" the entities. This will bind to the underlying entities under management to update the each [entity](#)'s sensors (e.g. to poll over HTTP or JMX). This new state will be reported in the web-console and can also trigger any registered policies.

Handling Rebind Failures

If rebind fails for any reason, details of the underlying failures will be reported in the `brooklyn.debug.log`. This will include the entities, locations or policies which caused an issue, and in what way it failed. There are several approaches to resolving problems.

1) Determine Underlying Cause

Go through the log and identify the likely areas in the code from the error message.

2) Seek Help

Help can be found by contacting the Apache Brooklyn mailing list.

3) Fix-up the State

The state of each [entity](#), [location](#), [policy](#) and [enricher](#) is persisted in XML. It is thus human readable and editable.

After first taking a backup of the state, it is possible to modify the state. For example, an offending [entity](#) could be removed, or references to that [entity](#) removed, or its XML could be fixed to remove the problem.

4) Fixing with Groovy Scripts

The final (powerful and dangerous!) tool is to execute Groovy code on the running Brooklyn instance. If authorized, the REST api allows arbitrary Groovy scripts to be passed in and executed. This allows the state of entities to be modified (and thus fixed) at runtime.

If used, it is strongly recommended that Groovy scripts are run against a disconnected Brooklyn instance. After fixing the entities, locations and/or policies, the Brooklyn instance's new persisted state can be copied and used to fix the production instance.

Writing Persistable Code

The most common problem on rebind is that custom `entity` code has not been written in a way that can be persisted and/or rebound.

The rule of thumb when implementing new entities, locations, policies and enrichers is that all state must be persistable. All state must be stored as config or as attributes, and must be serializable. For making backwards compatibility simpler, the persisted state should be clean.

Below are tips and best practices for when implementing an `entity` in Java (or any other JVM language).

How to store `entity` state:

- Config keys and values are persisted.
- Store an `entity`'s runtime state as attributes.
- Don't store state in arbitrary fields - the field will not be persisted (this is a design decision, because Brooklyn cannot intercept the field being written to, so cannot know when to persist).
- Don't just modify the retrieved attribute value (e.g. `getAttribute(MY_LIST).add("a")` is bad). The value may not be persisted unless `setAttribute()` is called.
- For special cases, it is possible to call `entity.requestPersist()` which will trigger asynchronous persistence of the `entity`.
- Overriding (and customizing) of `getRebindSupport()` is discouraged - this will change in a future version.

How to store `policy/enricher/location` state:

- Store values as config keys where applicable.
- Unfortunately these (currently) do not have attributes. Normally the state of a `policy` or `enricher` is transient - on rebind it starts afresh, for example with monitoring the performance or health metrics rather than relying on the persisted values.
- For special cases, you can annotate a field with `@SetFromFlag` for it be persisted. When you call `requestPersist()` then values of these fields will be scheduled to be persisted. *Warning: the `@SetFromFlag` functionality may change in future versions.*

Persistable state:

- Ensure values can be serialized. This (currently) uses xstream, which means it does not need to implement `Serializable`.
- Always use static (or top-level) classes. Otherwise it will try to also persist the outer instance!
- Any reference to an `entity` or `location` will be automatically swapped out for marker, and re-injected with the new `entity/location` instance on rebind. The same applies for policies, enrichers, feeds, catalog items and `ManagementContext`.

Behaviour on rebind:

- By extending `SoftwareProcess`, entities get a lot of the rebind logic for free. For example, the default `rebind()` method will call `connectSensors()`. See [SoftwareProcess Lifecycle](#) for more details.
- If necessary, implement rebind. The `entity.rebind()` is called automatically by the Brooklyn framework on rebind,

after configuring the `entity`'s config/attributes but before the `entity` is managed. Note that `init()` will not be called on rebind.

- Feeds will be persisted if and only if `entity.addFeed(...)` was called. Otherwise the feed needs to be re-registered on rebind. *Warning: this behaviour may change in future version.*
- All functions/predicates used with persisted feeds must themselves be persistable - use of anonymous inner classes is strongly discouraged.
- Subscriptions (e.g. from calls to `subscribe(...)` for `sensor` events) are not persisted. They must be re-registered on rebind. *Warning: this behaviour may change in future version.*

Below are tips to make backwards-compatibility easier for persisted state:

- Never use anonymous inner classes - even in static contexts. The auto-generated class names are brittle, making backwards compatibility harder.
- Always use sensible field names (and use `transient` whenever you don't want it persisted). The field names are part of the persisted state.
- Consider using Value Objects for persisted values. This can give clearer separation of responsibilities in your code, and clearer control of what fields are being persisted.
- Consider writing transformers to handle backwards-incompatible code changes. Brooklyn supports applying transformations to the persisted state, which can be done as part of an upgrade process.

Persisted State Backup

File system backup

When using the file system it is important to ensure it is backed up regularly.

One could use `rsync` to regularly backup the contents to another server.

It is also recommended to periodically create a complete archive of the state. A simple mechanism is to run a CRON job periodically (e.g. every 30 minutes) that creates an archive of the persistence directory, and uploads that to a backup facility (e.g. to S3).

Optionally, to avoid excessive load on the Brooklyn server, the archive-generation could be done on another "data" server. This could get a copy of the data via an `rsync` job.

An example script to be invoked by CRON is shown below:

```
DATE=`date "+%Y%m%d.%H%M.%S"`
BACKUP_FILENAME=/path/to/archives/back-{$DATE}.tar.gz
DATA_DIR=/path/to/base/dir/data

tar --exclude '*/backups/*' -czvf $BACKUP_FILENAME $DATA_DIR
# For s3cmd installation see http://s3tools.org/repositories
s3cmd put $BACKUP_FILENAME s3://mybackupbucket
rm $BACKUP_FILENAME
```

Object store backup

Object Stores will normally handle replication. However, many such object stores do not handle versioning (i.e. to allow access to an old version, if an object has been incorrectly changed or deleted).

The state can be downloaded periodically from the object store, archived and backed up.

An example script to be invoked by CRON is shown below:

```
DATE=`date "+%Y%m%d.%H%M.%S"`
BACKUP_FILENAME=/path/to/archives/back-${DATE}.tar.gz
TEMP_DATA_DIR=/path/to/tempdir

brooklyn copy-state \
    --persistenceLocation named:my-persistence-location \
    --persistenceDir /path/to/bucket \
    --destinationDir $TEMP_DATA_DIR

tar --exclude '*/backups/*' -czvf $BACKUP_FILENAME $TEMP_DATA_DIR
# For s3cmd installation see http://s3tools.org/repositories
s3cmd put $BACKUP_FILENAME s3://mybackupbucket
rm $BACKUP_FILENAME
rm -r $TEMP_DATA_DIR
```

High Availability

Brooklyn will automatically run in HA mode if multiple Brooklyn instances are started pointing at the same persistence store. One Brooklyn node (e.g. the first one started) is elected as HA master: all *write operations* against Brooklyn entities, such as creating an application or invoking an *effector*, should be directed to the master.

Once one node is running as `MASTER`, other nodes start in either `STANDBY` or `HOT_STANDBY` mode:

- In `STANDBY` mode, a Brooklyn instance will monitor the master and will be a candidate to become `MASTER` should the master fail. Standby nodes do *not* attempt to rebind until they are elected master, so the state of existing entities is not available at the standby node. However a standby server consumes very little resource until it is promoted.
- In `HOT_STANDBY` mode, a Brooklyn instance will read and make available the live state of entities. Thus a hot-standby node is available as a read-only copy. As with the standby node, if a hot-standby node detects that the master fails, it will be a candidate for promotion to master.
- In `HOT_BACKUP` mode, a Brooklyn instance will read and make available the live state of entities, as a read-only copy. However this node is not able to become master, so it can safely be used to test compatibility across different versions.

To explicitly specify what HA mode a node should be in, the following options are available for the config option

`highAvailabilityMode` in `org.apache.brooklyn.osgilauncher.cfg`:

- `DISABLED`: management node works in isolation; it will not cooperate with any other standby/master nodes in management plane
- `AUTO`: will look for other management nodes, and will allocate itself as standby or master based on other nodes' states
- `MASTER`: will startup as master; if there is already a master then fails immediately
- `STANDBY`: will start up as lukewarm standby; if there is not already a master then fails immediately
- `HOT_STANDBY`: will start up as hot standby; if there is not already a master then fails immediately
- `HOT_BACKUP`: will start up as hot backup; this can be done even if there is not already a master; this node will not be a master

The REST API offers live detection and control of the HA mode, including setting priority to control which nodes will be promoted on master failure:

- `/server/ha/state`: Returns the HA state of a management node (GET), or changes the state (POST)
- `/server/ha/states`: Returns the HA states and detail for all nodes in a management plane
- `/server/ha/priority`: Returns the HA node priority for MASTER failover (GET), or sets that priority (POST)

Note that when POSTing to a non-master server it is necessary to pass a `Brooklyn-Allow-Non-Master-Access: true` header. For example, the following cURL command could be used to change the state of a `STANDBY` node on

`localhost:8082` to `HOT_STANDBY`:

```
curl -v -X POST -d mode=HOT_STANDBY -H "Brooklyn-Allow-Non-Master-Access: true" http://localhost:8082/v1/server/ha/state
```

Configuring HA - an example

This supplements the [High Availability](#) documentation and provides an example of how to configure a pair of Apache Brooklyn servers to run in master-standby mode with a shared NFS datastore

Prerequisites

- Two VMs (or physical machines) have been provisioned
- NFS or another suitable file system has been configured and is available to both VMs*
- An NFS folder has been mounted on both VMs at `/mnt/brooklyn-persistence` and both machines can write to the folder

* Brooklyn can be configured to use either an object store such as S3, or a shared NFS mount. The recommended option is to use an object store as described in the [Object Store Persistence](#) documentation. For simplicity, a shared NFS folder is assumed in this example

Launching

To start, download and install the latest Apache Brooklyn release on both VMs following the instructions in [Running Apache Brooklyn](#)

On the first VM, which will be the master node, set the following configuration options in

`org.apache.brooklyn.osgilauncher.cfg` :

- `highAvailabilityMode`: MASTER
- `persistMode`: AUTO
- `persistenceDir`: `/mnt/brooklyn-persistence`

Then launch Brooklyn with:

```
$ bin/start
```

If you are using RPMs/deb to install, please see the [Running Apache Brooklyn](#) documentation for the appropriate launch commands

Once Brooklyn has launched, on the second VM, set the following configuration options in

`org.apache.brooklyn.osgilauncher.cfg` :

- `highAvailabilityMode`: AUTO
- `persistMode`: AUTO
- `persistenceDir`: `/mnt/brooklyn-persistence`

Then launch the standby Brooklyn with:

```
$ bin/start
```

Failover

When running as a HA standby node, each standby Brooklyn server (in this case there is only one standby) will check the shared persisted state every one second to determine the state of the HA master. If no heartbeat has been recorded for 30 seconds, then an election will be performed and one of the standby nodes will be promoted to master. At this point all

requests should be directed to the new master node. If the master is terminated gracefully, the secondary will be immediately promoted to master. Otherwise, the secondary will be promoted after heartbeats are missed for a given length of time. This defaults to 30 seconds, and is configured in `brooklyn.cfg` using `brooklyn.ha.heartbeatTimeout`

In the event that tasks - such as the provisioning of a new `entity` - are running when a failover occurs, the new master will display the current state of the `entity`, but will not resume its provisioning or re-run any partially completed tasks. In this case it may be necessary to remove the `entity` and reprovision it. In the case of a failover whilst executing a task called by an `effector`, it may be possible to simply call the `effector` again

Client Configuration

It is the responsibility of the client to connect to the master Brooklyn server. This can be accomplished in a variety of ways:

• Reverse Proxy

To allow the client application to automatically fail over in the event of a master server becoming unavailable, or the promotion of a new master, a reverse proxy can be configured to route traffic depending on the response returned by `https://<ip-address>:8443/v1/server/ha/state` (see above). If a server returns `"MASTER"`, then traffic should be routed to that server, otherwise it should not be. The client software should be configured to connect to the reverse proxy server and no action is required by the client in the event of a failover. It can take up to 30 seconds for the standby to be promoted, so the reverse proxy should retry for at least this period, or the failover time should be reconfigured to be shorter

• Re-allocating an Elastic IP on Failover

If the cloud provider you are using supports Elastic or Floating IPs, then the IP address should be allocated to the HA master, and the client application configured to connect to the floating IP address. In the event of a failure of the master node, the standby node will automatically be promoted to master, and the floating IP will need to be manually re-allocated to the new master node. No action is required by the client in the event of a failover. It is possible to automate the re-allocation of the floating IP if the Brooklyn servers are deployed and managed by Brooklyn using the `entity org.apache.brooklyn.entity.brooklynnode.BrooklynCluster`

• Client-based failover

In this scenario, the responsibility for determining the Brooklyn master server falls on the client application. When configuring the client application, a list of all servers in the cluster is passed in at application startup. On first connection, the client application connects to any of the members of the cluster to retrieve the HA states (see above). The JSON object returned is used to determine the addresses of all members of the cluster, and also to determine which node is the HA master

In the event of a failure of the master node, the client application should then retrieve the HA states of the cluster from any of the other cluster members. This is the same process as when the application first connects to the cluster. The client should refresh its list of cluster members and determine which node is the HA master

It is also recommended that the client application periodically checks the status of the cluster and updates its list of addresses. This will ensure that failover is still possible if the standby server(s) has been replaced. It also allows additional standby servers to be added at any time

Testing

You can confirm that Brooklyn is running in high availability mode on the master by logging into the web console at `https://<ip-address>:8443` . Similarly you can log into the web console on the standby VM where you will see a warning that the server is not the high availability master.

To test a failover, you can simply terminate the process on the first VM and log into the web console on the second VM. Upon launch, Brooklyn will output its PID to the file `pid.txt` ; you can force an immediate (non-graceful) termination of the process by running the following command from the same directory from which you launched Brooklyn:

```
$ kill -9 $(cat pid.txt)
```

It is also possible to check the high availability state of a running Brooklyn server using the following curl command:

```
$ curl -k -u myusername:mypassword https://<ip-address>:8443/v1/server/ha/state
```

This will return one of the following states:

```
"INITIALIZING"  
"STANDBY"  
"HOT_STANDBY"  
"HOT_BACKUP"  
"MASTER"  
"FAILED"  
"TERMINATED"
```

Note: The quotation characters will be included in the reply

To obtain information about all of the nodes in the cluster, run the following command against any of the nodes in the cluster:

```
$ curl -k -u myusername:mypassword https://<ip-address>:8443/v1/server/ha/states
```

This will return a JSON document describing the Brooklyn nodes in the cluster. An example of two HA Brooklyn nodes is as follows (whitespace formatting has been added for clarity):

```
{  
  ownId: "XkJeXUXE",  
  masterId: "yAVz0fzo",  
  nodes: {  
    yAVz0fzo: {  
      nodeId: "yAVz0fzo",  
      nodeUri: "https://<server1-ip-address>:8443/",  
      status: "MASTER",  
      localTimestamp: 1466414301065,  
      remoteTimestamp: 1466414301000  
    },  
    XkJeXUXE: {  
      nodeId: "XkJeXUXE",  
      nodeUri: "https://<server2-ip-address>:8443/",  
      status: "STANDBY",  
      localTimestamp: 1466414301066,  
      remoteTimestamp: 1466414301000  
    }  
  },  
  links: { }  
}
```


The examples above show how to use `curl` to manually check the status of Brooklyn via its REST API. The same REST API calls can also be used by automated third party monitoring tools such as Nagios

Logging

Brooklyn uses the SLF4J logging facade, which allows use of many popular frameworks including `logback`, `java.util.logging` and `log4j`.

The convention for log levels is as follows:

- `ERROR` and above: exceptional situations which indicate that something has unexpectedly failed or some other problem has occurred which the user is expected to attend to
- `WARN`: exceptional situations which the user may wish to know about but which do not necessarily indicate failure or require a response
- `INFO`: a synopsis of activity, but which should not generate large volumes of events nor overwhelm a human observer
- `DEBUG` and lower: detail of activity which is not normally of interest, but which might merit closer inspection under certain circumstances.

Loggers follow the `package.ClassName` naming standard.

Using Logback through OSGi Pax Logging

In the OSGi based Apache Brooklyn logging is configured from ops4j pax logging.

See: [Logging - OSGi based Apache Brooklyn](https://ops4j1.jira.com/wiki/display/paxlogging/Configuration)
<https://ops4j1.jira.com/wiki/display/paxlogging/Configuration>

Standard Configuration

A `logback.xml` file is included in the `conf/` directory of the Brooklyn distro; this is read by `brooklyn` at launch time. Changes to the logging configuration, such as new appenders or different log levels, can be made directly in this file or in a new file included from this.

Advanced Configuration

The default `logback.xml` file references a collection of other log configuration files included in the Brooklyn jars. It is necessary to understand the source structure in the [logback-includes](#) project.

For example, to change the debug log inclusions, create a folder `brooklyn` under `conf` and create a file `logback-debug.xml` based on the [brooklyn/logback-debug.xml](#) from that project.

Log File Backup

This sub-section is a work in progress; feedback from the community is extremely welcome.

The default rolling log files can be backed up periodically, e.g. using a CRON job.

Note however that the rolling log file naming scheme will rename the historic zipped log files such that `brooklyn.debug-1.log.zip` is the most recent zipped log file. When the current `brooklyn.debug.log` is to be zipped, the previous zip file will be renamed `brooklyn.debug-2.log.zip`. This renaming of files can make RSYNC or backups tricky.

An option is to covert/move the file to a name that includes the last-modified timestamp. For example (on mac):

```
LOG_FILE=brooklyn.debug-1.log.zip
TIMESTAMP=`stat -f '%Um' $LOG_FILE`
mv $LOG_FILE /path/to/archive/brooklyn.debug-$TIMESTAMP.log.zip
```

Logging aggregators

Integration with systems like Logstash and Splunk is possible using standard logback configuration. Logback can be configured to [write to the syslog](#), which can then [feed its logs to Logstash](#).

For More Information

The following resources may be useful when configuring logging:

- The [logback-includes](#) project
- [Brooklyn Developer Guide](#) logging tips
- The [Logback Project](#) home page

Externalized Configuration

Sometimes it is useful that configuration in a [blueprint](#), or in Brooklyn itself, is not given explicitly, but is instead replaced with a reference to some other storage system. For example, it is undesirable for a [blueprint](#) to contain a plain-text password for a production system, especially if (as we often recommend) the blueprints are kept in the developer's source code control system.

To handle this problem, Apache Brooklyn supports externalized configuration. This allows a [blueprint](#) to refer to a piece of information that is stored elsewhere. `brooklyn.cfg` defines the external suppliers of configuration information. At runtime, when Brooklyn finds a reference to externalized configuration in a [blueprint](#), it consults `brooklyn.cfg` for information about the supplier, and then requests that the supplier return the information required by the [blueprint](#).

Take, as a simple example, a web app which connects to a database. In development, the developer is running a local instance of PostgreSQL with a simple username and password. But in production, an enterprise-grade cluster of PostgreSQL is used, and a dedicated service is used to provide passwords. The same [blueprint](#) can be used to service both groups of users, with `brooklyn.cfg` changing the behaviour depending on the deployment environment.

Here is the [blueprint](#):

```
name: MyApplication
services:
- type: brooklyn.entity.webapp.jboss.JBoss7Server
  name: AppServer HelloWorld
  brooklyn.config:
    wars.root: http://search.maven.org/remotecontent?filepath=org/apache/brooklyn/example/brooklyn-example-hello-world-sql-webapp/0.8.0-incubating/brooklyn-example-hello-world-sql-webapp-0.8.0-incubating.war
    http.port: 8080+
    java.sysprops:
      brooklyn.example.db.url:
        $brooklyn:formatString:
          - jdbc:postgresql://%s/myapddb?user=%s\\&password=%s
          - $brooklyn:external("servers", "postgresql")
          - $brooklyn:external("credentials", "postgresql-user")
          - $brooklyn:external("credentials", "postgresql-password")
```

You can see that when we are building up the JDBC URL, we are using the `external` function. This takes two parameters: the first is the name of the configuration supplier, the second is the name of a key that is stored by the configuration supplier. In this case we are using two different suppliers: `servers` to store the [location](#) of the server, and `credentials` which is a security-optimized supplier of secrets.

Developers would add lines like this to the `brooklyn.cfg` file on their workstation:

```
brooklyn.external.servers=org.apache.brooklyn.core.config.external.InPlaceExternalConfigSupplier
brooklyn.external.servers.postgresql=127.0.0.1
brooklyn.external.credentials=org.apache.brooklyn.core.config.external.InPlaceExternalConfigSupplier
brooklyn.external.credentials.postgresql-user=admin
brooklyn.external.credentials.postgresql-password=admin
```

In this case, all of the required information is included in-line in the local `brooklyn.cfg`.

Whereas in production, `brooklyn.cfg` might look like this:

```
brooklyn.external.servers=org.apache.brooklyn.core.config.external.PropertiesFileExternalConfigSupplier
brooklyn.external.servers.propertiesUrl=https://ops.example.com/servers.properties
brooklyn.external.credentials=org.apache.brooklyn.core.config.external.vault.VaultAppIdExternalConfigSupplier
brooklyn.external.credentials.endpoint=https://vault.example.com
brooklyn.external.credentials.path=secret/enterprise-postgres
```

```
brooklyn.external.credentials.appId=MyApp
```

In this case, the list of servers is stored in a properties file located on an Operations Department web server, and the credentials are stored in an instance of [Vault](#). More information on these providers is below.

For demo purposes, there is a pre-defined external provider called `brooklyn-demo-sample` which defines `hidden-brooklyn-password` as `br00k11n`. This is used in some of the sample blueprints, referencing `$brooklyn:external("brooklyn-demo-sample", "hidden-brooklyn-password")`. The value used here can be overridden with the following in your `brooklyn.cfg`:

```
brooklyn.external.brooklyn-demo-sample=org.apache.brooklyn.core.config.external.InPlaceExternalConfigSupplier
brooklyn.external.brooklyn-demo-sample.hidden-brooklyn-password=new_password
```

Defining Suppliers

External configuration suppliers are defined in `brooklyn.cfg`. The minimal definition is of the form:

```
brooklyn.external.supplierName = className
```

This defines a supplier named *supplierName*. Brooklyn will attempt to instantiate *className*; it is this class which will provide the behaviour of how to retrieve data from the supplier. Brooklyn includes a number of supplier implementations; see below for more details.

Suppliers may require additional configuration options. These are given as additional properties in `brooklyn.cfg`:

```
brooklyn.external.supplierName = className
brooklyn.external.supplierName.firstConfig = value
brooklyn.external.supplierName.secondConfig = value
```

Referring to External Configuration in Blueprints

Externalized configuration adds a new function to the Brooklyn [blueprint](#) language DSL, `$brooklyn:external`. This function takes two parameters:

1. `supplier`
2. `key`

When resolving the external reference, Brooklyn will first identify the *supplier* of the information, then it will give the supplier the *key*. The returned value will be substituted into the [blueprint](#).

You can use `$brooklyn:external` directly:

```
name: MyApplication
brooklyn.config:
  example: $brooklyn:external("supplier", "key")
```

or embed the `external` function inside another `$brooklyn` DSL function, such as `$brooklyn:formatString`:

```
name: MyApplication
brooklyn.config:
  example: $brooklyn:formatString("%s", external("supplier", "key"))
```

Referring to External Configuration in `brooklyn.cfg`

The same `blueprint` language DSL can be used from `brooklyn.cfg`. For example:

```
brooklyn.location.jclouds.aws-ec2.identity=$brooklyn:external("mysupplier", "aws-identity")
brooklyn.location.jclouds.aws-ec2.credential=$brooklyn:external("mysupplier", "aws-credential")
```

Referring to External Configuration in Catalog Items

The same `blueprint` language DSL can be used within `YAML` catalog items. For example:

```
brooklyn.catalog:
  id: com.example.myblueprint
  version: "1.2.3"
  itemType: entity
  brooklyn.libraries:
    - >
      $brooklyn:formatString("https://%s:%s@repo.example.com/libs/myblueprint-1.2.3.jar",
        external("mysupplier", "username"), external("mysupplier", "password"))
  item:
    type: com.example.MyBlueprint
```

Note the `>` in the example above is used to split across multiple lines.

Suppliers available with Brooklyn

Brooklyn ships with a number of external configuration suppliers ready to use.

In-place

InPlaceExternalConfigSupplier embeds the configuration keys and values as properties inside `brooklyn.cfg`. For example:

```
brooklyn.external.servers=org.apache.brooklyn.core.config.external.InPlaceExternalConfigSupplier
brooklyn.external.servers.postgresql=127.0.0.1
```

Then, a `blueprint` which referred to `$brooklyn:external("servers", "postgresql")` would receive the value `127.0.0.1`.

Properties file

PropertiesFileExternalConfigSupplier loads a properties file from a URL, and uses the keys and values in this file to respond to configuration lookups.

Given this configuration:

```
brooklyn.external.servers=org.apache.brooklyn.core.config.external.PropertiesFileExternalConfigSupplier
brooklyn.external.servers.propertiesUrl=https://ops.example.com/servers.properties
```

This would cause the supplier to download the given URL. Assuming that the file contained this entry:

```
postgresql=127.0.0.1
```

Then, a `blueprint` which referred to `$brooklyn:external("servers", "postgresql")` would receive the value `127.0.0.1`.

Vault

`Vault` is a server-based tool for managing secrets. Brooklyn provides suppliers that are able to query the Vault REST API for configuration values. The different suppliers implement alternative authentication options that Vault provides.

For *all* of the authentication methods, you must always set these properties in `brooklyn.cfg`:

```
brooklyn.external.supplierName.endpoint=<Vault HTTP/HTTPS endpoint>
brooklyn.external.supplierName.path=<path to a Vault object>
```

For example, if the path is set to `secret/brooklyn`, then attempting to retrieve the key `foo` would cause Brooklyn to retrieve the value of the `foo` key on the `secret/brooklyn` object. This value can be set using the Vault CLI like this:

```
vault write secret/brooklyn foo=bar
```

Authentication by username and password

The `userpass` plugin for Vault allows authentication with username and password.

```
brooklyn.external.supplierName=org.apache.brooklyn.core.config.external.vault.VaultUserPassExternalConfigSupplier
brooklyn.external.supplierName.username=fred
brooklyn.external.supplierName.password=s3kr1t
```

Authentication using App ID

The `app_id` plugin for Vault allows you to specify an "app ID", and then designate particular "user IDs" to be part of the app. Typically the app ID would be known and shared, but user ID would be autogenerated on the client in some way. Brooklyn implements this by determining the MAC address of the server running Brooklyn (expressed as 12 lower case hexadecimal digits without separators) and passing this as the user ID.

```
brooklyn.external.supplierName=org.apache.brooklyn.core.config.external.vault.VaultAppIdExternalConfigSupplier
brooklyn.external.supplierName.appId=MyApp
```

If you do not wish to use the MAC address as the user ID, you can override it with your own choice of user ID:

```
brooklyn.external.supplierName.userId=server3.cluster2.europe
```

Authentication by fixed token

If you have a fixed token string, then you can use the `VaultTokenExternalConfigSupplier` class and provide the token in

`brooklyn.cfg`:

```
brooklyn.external.supplierName=org.apache.brooklyn.core.config.external.vault.VaultTokenExternalConfigSupplier
brooklyn.external.supplierName.token=1091fc84-70c1-b266-b99f-781684dd0d2b
```

This supplier is suitable for "smoke testing" the Vault supplier using the Initial Root Token or similar. However it is not suitable for production use as it is inherently insecure - should the token be compromised, an attacker could have complete access to your Vault, and the cleanup operation would be difficult. Instead you should use one of the other suppliers.

Writing Custom External Configuration Suppliers

Supplier implementations must conform to the `brooklyn.config.external.ExternalConfigSupplier` interface, which is very simple:

```
String getName();  
String get(String key);
```

Classes implementing this interface can be placed in the `lib/dropins` folder of Brooklyn, and then the supplier defined in `brooklyn.cfg` as normal.

Requirements

Server Specification

The size of server required by Brooklyn depends on the amount of activity. This includes:

- the number of entities/VMs being managed
- the number of VMs being deployed concurrently
- the amount of management and monitoring required per [entity](#)

For dev/test or when there are only a handful of VMs being managed, a small VM is sufficient. For example, an AWS m3.medium with one vCPU, 3.75GiB RAM and 4GB disk.

For larger production uses, a more appropriate machine spec would be two or more cores, at least 8GB RAM and 100GB disk. The disk is just for logs, a small amount of persisted state, and any binaries for custom blueprints/integrations.

Disk Space

There are three main consumers of disk space:

- **Static files:** these are the Apache Brooklyn distribution with its own dependencies, OSGi bundles for custom blueprints and integrations installed to the `deploy/` directory, plus `data/` directory which is generated on first launch. Note that Brooklyn requires that Java is installed which you may have to consider when calculating disk space requirements.
- **Persisted state:** when using [Persistence](#) -- which is a prerequisite for [High Availability](#) -- Brooklyn will save data to a store [location](#). Items in the persisted state include metadata about the Brooklyn servers, catalog items, and metadata about all running applications and entities.
- **Log files:** Brooklyn writes info and debug log files. By default, these are written to the local filesystem. This can be reconfigured to set the destination and to increase or decrease the detail in the logs. See the [Logging](#) section for more details.

The Apache Brooklyn distribution itself, when unpacked, consumes approximately 75MB of disk space. This includes everything needed to run Brooklyn except for a Java VM. The space consumed by additional binaries for custom blueprints and integrations is application-specific.

Persisted state, excluding catalog data, is relatively small, starting at approximately 300KB for a clean, idle Brooklyn server. Deploying blueprints will add to this - how much depends exactly on the entities involved and is therefore application specific, but as a guideline, a 3-node Riak cluster adds approximately 500KB to the persistence store.

Log data can be a large consumer of disk space. By default Brooklyn generates two logfiles, one which logs notable information only, and another which logs at a debug level. Each logfile rotates when it hits a size of 100MB; a maximum of 10 log files are retained for each type. The two logging streams combined, therefore, can consume up to 2GB of disk space.

In the default configuration of Brooklyn's `.tar.gz` and `.zip` distributions, logs are saved to the Brooklyn installation directory. You will most likely want to [reconfigure Brooklyn's logging](#) to save logs to a [location](#) elsewhere. In the `.rpm` and `.deb` packaging, logging files will be located under `/var/log`. You can further reconfiguring the logging detail level and log rotation according to your organisation's [policy](#).

OS Requirements

Brooklyn is tested against CentOS (6 or later), RHEL (6 or later), Ubuntu (14.04 or later), OS X, and Windows.

Software Requirements

Brooklyn requires Java 8 (JRE or JDK) or later. OpenJDK is recommended. Brooklyn has also been tested on the Oracle JVM and IBM J9.

Configuration Requirements

Ports

The ports used by Brooklyn are:

- 8443 for https, to expose the web-console and REST api.
- 8081 for http, to expose the web-console and REST api.

Whether to use https rather than http is configurable using the CLI option `--https`; the port to use is configurable using the CLI option `--port <port>`.

To enable remote Brooklyn access, ensure these ports are open in the firewall. For example, to open port 8443 in iptables, use the command:

```
/sbin/iptables -I INPUT -p TCP --dport 8443 -j ACCEPT
```

Locale

Brooklyn expects a sensible set of locale information and time zones to be available; without this, some time-and-date handling may be surprising.

Brooklyn parses and reports times according to the time zone set at the server. If Brooklyn is targetting geographically distributed users, it is normally recommended that the server's time zone be set to UTC.

User Setup

It is normally recommended that Brooklyn run as a non-root user with keys installed to `~/.ssh/id_rsa{, .pub}`.

Linux Kernel Entropy

Check that the `linux kernel entropy` is sufficient.

System Resource Limits

Check that the `ulimit values` are sufficiently high.

Upgrade

This guide provides all necessary information to upgrade Apache Brooklyn for both the RPM/DEB and Tarball packages.

Backwards Compatibility

Apache Brooklyn version 0.12.0 onward runs primarily inside a Karaf container. When upgrading from 0.11.0 or below, this update changes the mechanisms for launching Brooklyn. This will impact any custom scripting around the launching of Brooklyn, and the supplying of command line arguments.

Use of the `lib/dropins` and `lib/patch` folders will no longer work (because Karaf does not support that kind of classloading). Instead, code must be built and installed as [OSGi bundles](#).

Upgrading

- Use of RPM and DEB is now recommended where possible, rather than the tar.gz. This entirely replaces the previous install.
- CentOS 7.x is recommended over CentOS 6.x (note: the RPM **will not work** on CentOS 6.x)

Upgrade from Apache Brooklyn 0.12.0 onward

Upgrading an RPM or DEB package

1. **Important!** Backup persisted state and custom configuration, in case you need to rollback to a previous version.
 - i. By default, persisted state is located at `/var/lib/brooklyn`. The `persistenceDir` and `persistenceLocation` are configured in the file `/etc/brooklyn/org.apache.brooklyn.osgilauncher.cfg`. The persistence details will be logged in `/var/log/brooklyn/brooklyn.info.log` at startup time.
 - ii. Configuration files are in `/etc/brooklyn`.
2. Upgrade Apache Brooklyn:
 - i. [Download](#) the new RPM/DEB package
 - ii. Upgrade Apache Brooklyn:

```
# CentOS / RHEL
sudo yum upgrade apache-brooklyn-xxxx.noarch.rpm

# Ubuntu / Debian
sudo dpkg -i apache-brooklyn-xxxx.deb
```

If there are conflicts in configuration files (located in `/etc/brooklyn`), the upgrade will behave differently based on the package you are using:

- RPM: the upgrade will keep the previously installed one and save the new version, with the suffix `.rpmsave`. You will then need to check and manually resolve those.
- DEB: the upgrade will ask you what to do.

3. Start Apache Brooklyn:

```
# CentOS 7 / RHEL
sudo systemctl start brooklyn
# CentOS 6 and older
sudo initctl start brooklyn

# Ubuntu / Debian
start brooklyn
```

Wait for Brooklyn to be running (i.e. its web-console is responsive)

Upgrading using a .tar.gz archive

1. Stop Apache Brooklyn:

```
./bin/stop brooklyn
```

If this does not stop it within a few seconds (as checked with `sudo ps aux | grep karaf`), then use `sudo kill <JAVA_PID>`

2. **Important!** Backup persisted state and custom configuration.

- i. By default, persisted state is located at `~/.brooklyn/brooklyn-persisted-state`. The `persistenceDir` and `persistenceLocation` are configured in the file `./etc/org.apache.brooklyn.osgilauncher.cfg`. The persistence details will be logged in `./log/brooklyn.info.log` at startup time.
- ii. Configuration files are in `./etc/`. Any changes to these configuration files will need to be re-applied after reinstalling Brooklyn.

3. Install new version of Apache Brooklyn:

- i. [Download](#) the new tarball zip package.
- ii. Install Brooklyn:

```
tar -zxf apache-brooklyn-xxxx.tar.gz
cd apache-brooklyn-xxxx
```

4. Restore any changes to the configuration files (see step 2).

5. Validate that the new release works, by starting in "HOT_BACKUP" mode.

- i. Before starting Brooklyn, reconfigure `./etc/org.apache.brooklyn.osgilauncher.cfg` and set `highAvailabilityMode=HOT_BACKUP`. This way when Brooklyn is started, it will only read and validate the persisted state and will not write into it.
- ii. Start Apache Brooklyn:

```
./bin/start brooklyn
```

- iii. Check whether you have rebind ERROR messages in `./log/brooklyn.info.log`, e.g. `sudo grep -E "WARN|ERROR" /opt/brooklyn/log/brooklyn.debug.log`. If you do not have such errors you can proceed.
- iv. Stop Apache Brooklyn:

```
./bin/stop brooklyn
```

- v. Change the `highAvailabilityMode` to the default (AUTO) by commenting it out in `./etc/org.apache.brooklyn.osgilauncher.cfg`.

6. Start Apache Brooklyn:

```
./bin/start brooklyn
```

Wait for Brooklyn to be running (i.e. its web-console is responsive).

7. Update the catalog, using the `br` command:

- i. [Download](#) the `br` tool.
- ii. Login with `br`: `br login http://localhost:8081 <user> <password>`.
- iii. Update the catalog: `br catalog add /opt/brooklyn/catalog/catalog.bom`.

Upgrade from Apache Brooklyn 0.11.0 and below

Upgrading an RPM or DEB package

1. Stop Apache Brooklyn:

```
# CentOS 7 / RHEL
sudo systemctl stop brooklyn
# CentOS6 and older
sudo initctl stop brooklyn

# Ubuntu / Debian
stop brooklyn
```

If this does not stop it within a few seconds (as checked with `sudo ps aux | grep brooklyn`), then use `sudo kill <JAVA_PID>`.

2. **Important!** Backup persisted state and custom configuration.

- i. By default, persisted state is located at `/opt/brooklyn/.brooklyn/..`. The `persistenceDir` and `persistenceLocation` are configured in the file `./etc/org.apache.brooklyn.osgilauncher.cfg`. The persistence details will be logged in `./log/brooklyn.info.log` at startup time.
- ii. Configuration files are in `./etc/`. Any changes to these configuration files will need to be re-applied after reinstalling Brooklyn.

3. Delete the existing Apache Brooklyn install:

- i. Remove Brooklyn package:

```
# CentOS / RHEL
sudo yum erase apache-brooklyn

# Ubuntu / Debian
sudo dpkg -r apache-brooklyn
```

- i. On CentOS 7 run `sudo systemctl daemon-reload`.
- ii. Confirm that Brooklyn is definitely not running (see step 1 above).

- iii. Delete the Brooklyn install directory: `sudo rm -r /opt/brooklyn` as well as the Brooklyn log directory: `sudo rm -r /var/log/brooklyn/`
- 4. Make sure you have Java 8. By default CentOS images come with JRE6 which is incompatible version for Brooklyn. If CentOS is prior to 6.8 upgrade nss: `yum -y upgrade nss`

- 5. Install new version of Apache Brooklyn:

- i. [Download](#) the new RPM/DEB package.
- ii. Install Apache Brooklyn:

```
# CentOS / RHEL
sudo yum install apache-brooklyn-xxxx.noarch.rpm

# Ubuntu / Debian
sudo dpkg -i apache-brooklyn-xxxx.deb
```

- 6. Restore the persisted state and configuration.

- i. Stop the Brooklyn service:

```
# CentOS 7 / RHEL
sudo systemctl stop brooklyn
# CentOS 6 and older
sudo initctl stop brooklyn

# Ubuntu / Debian
stop brooklyn
```

Confirm that Brooklyn is no longer running (see step 1).

- ii. Restore the persisted state directory into `/var/lib/brooklyn` and any changes to the configuration files (see step 2). Ensure owner/permissions are correct for the persisted state directory, e.g.: `sudo chown -r brooklyn:brooklyn /var/lib/brooklyn`
- 7. Validate that the new release works, by starting in "HOT_BACKUP" mode.
 - i. Before starting Brooklyn, reconfigure `/etc/brooklyn/org.apache.brooklyn.osgilauncher.cfg` and set `highAvailabilityMode=HOT_BACKUP`. This way when Brooklyn is started, it will only read and validate the persisted state and will not write into it.
 - ii. Start Apache Brooklyn:

```
# CentOS 7 / RHEL
sudo systemctl start brooklyn
# CentOS 6 and older
sudo initctl start brooklyn

# Ubuntu / Debian
start brooklyn
```

- iii. Check whether you have rebind ERROR messages in the Brooklyn logs, e.g. `sudo grep -E "Rebind|WARN|ERROR" /var/log/brooklyn/brooklyn.debug.log`. If you do not have such errors you can proceed.
- iv. Stop Brooklyn:

```
# CentOS 7 / RHEL
sudo systemctl stop brooklyn
# CentOS 6 and older
sudo initctl stop brooklyn
```

```
# Ubuntu / Debian
stop brooklyn
```

- v. Change the `highAvailabilityMode` to the default (AUTO) by commenting it out in `./etc/org.apache.brooklyn.osgilauncher.cfg`.

8. Start Apache Brooklyn:

```
# CentOS 7 / RHEL
sudo systemctl start brooklyn
# CentOS 6 and older
sudo initctl start brooklyn

# Ubuntu / Debian
start brooklyn
```

Wait for Brooklyn to be running (i.e. its web-console is responsive).

9. Update the catalog, using the `br` command:
- Download the `br` tool (i.e. from the "CLI Download" link in the web-console).
 - Login with `br`: `br login http://localhost:8081 <user> <password>`.
 - Update the catalog: `br catalog add /opt/brooklyn/catalog/catalog.bom`.

Upgrading using a .tar.gz archive

Same instructions as above.

Rollback

This section applies only with you are using the RPM/DEB packages. To perform a rollback, please follow these instructions:

```
# CentOS / RHEL
yum downgrade apache-brooklyn.noarch

# Ubuntu Debian
dpkg -i apache-brooklyn-xxxx.deb
```

Note that to downgrade a DEB package is essentially installing a previous version therefore you need to [download](#) the version you want to downgrade to before hand.

How to stop your service

On systemd:

```
systemctl stop brooklyn
```

On upstart:

```
stop brooklyn
```


Web login credentials

- User credentials should now be recorded in `brooklyn.cfg`.
- Brooklyn will still read them from both `brooklyn.cfg` and `~/.brooklyn/brooklyn.properties`.
- Configure a username/password by modifying `brooklyn.cfg`. An example entry is:

```
brooklyn.webconsole.security.users=admin  
brooklyn.webconsole.security.user.admin.password=password2
```

Persistence

If you have persisted state you wish to rebind to, persistence is now configured in the following files:

- `brooklyn.cfg`
- `org.apache.brooklyn.osgilauncher.cfg`

For example, to use S3 for the persisted state, add the following to `brooklyn.cfg`:

```
brooklyn.location.named.aws-s3-eu-west-1:aws-s3:eu-west-1  
brooklyn.location.named.aws-s3-eu-west-1.identity=<ADD_CREDS>  
brooklyn.location.named.aws-s3-eu-west-1.credential=<ADD_CREDS>
```

To continue the S3 example, for the persisted state, add the following to `org.apache.brooklyn.osgilauncher.cfg`:

```
persistenceLocation=aws-s3-eu-west-1  
persistenceDir=<ADD_HERE>
```

Apache Brooklyn should be stopped before this file is modified, and then restarted with the new configuration.

Note that you can not store the credentials (for e.g. aws-s3-eu-west-1) in the catalog because that catalog is stored in the persisted state. Apache Brooklyn needs to know it in order to read the persisted state at startup time.

If binding to existing persisted state, an additional command is required to update the existing catalog with the Brooklyn 0.12.0 versions. Assuming Brooklyn has been installed to `/opt/brooklyn` (as is done by the RPM and DEB):

```
br catalog add /opt/brooklyn/catalog/catalog.bom
```

All existing custom jars previously added to lib/plugins (e.g. for Java-based entities) need to be converted to OSGi bundles, and installed in Karaf. The use of the "brooklyn.libraries" section in catalog.bom files will continue to work.

Security Guidelines

Brooklyn Server

Web-console and REST api

Users are strongly encouraged to use HTTPS, rather than HTTP.

The use of LDAP is encouraged, rather than basic auth.

Configuration of "entitlements" is encouraged, to lock down access to the REST api for different users.

Brooklyn user

Users are strongly discouraged from running Brooklyn as root.

For production use-cases (i.e. where Brooklyn will never deploy to "localhost"), the user under which Brooklyn is running should not have `sudo` rights.

Persisted State

Use of an object store is recommended (e.g. using S3 compliant or Swift API) - thus making use of the security features offered by the chosen object store.

File-based persistence is also supported. Permissions of the files will automatically be 600 (i.e. read-write only by the owner). Care should be taken for permissions of the relevant mount points, disks and directories.

Credential Storage

For credential storage, users are strongly encouraged to consider using the "externalised configuration" feature. This allows credentials to be retrieved from a store managed by you, rather than being stored within `YAML` blueprints or `brooklyn.cfg`.

A secure credential store is strongly recommended, such as use of [HashiCorp's Vault](#) - see `org.apache.brooklyn.core.config.external.vault.VaultExternalConfigSupplier`.

Infrastructure Access

Cloud Credentials and Access

Users are strongly encouraged to create separate cloud credentials for Brooklyn's API access.

Users are also encouraged to (where possible) configure the cloud provider for only minimal API access (e.g. using AWS IAM).

VM Image Credentials

Users are strongly discouraged from using hard-coded passwords within VM images. Most cloud providers/APIs provide a mechanism to instead set an auto-generated password or to create an entry in `~/.ssh/authorized_keys` (prior to the VM being returned by the cloud provider).

If a hard-coded credential is used, then Brooklyn can be configured with this "loginUser" and "loginUser.password" (or "loginUser.privateKeyData"), and can change the password and disable root login.

VM Users

It is strongly discouraged to use the root user on VMs being created or managed by Brooklyn. SSH-ing on the VM should be done on rare cases such as initial Apache Brooklyn setup, Apache Brooklyn upgrade and other important maintenance occasions.

SSH keys

Users are strongly encouraged to use SSH keys for VM access, rather than passwords.

This SSH key could be a file on the Brooklyn server. However, a better solution is to use the "externalised configuration" to return the "privateKeyData". This better supports upgrading of credentials.

Install Artifact Downloads

When Brooklyn executes scripts on remote VMs to install software, it often requires downloading the install artifacts. For example, this could be from an RPM repository or to retrieve `.zip` installers.

By default, the RPM repositories will be whatever the VM image is configured with. For artifacts to be downloaded directly, these often default to the public site (or mirror) for that software product.

Where users have a private RPM repository, it is strongly encouraged to ensure the VMs are configured to point at this.

For other artifacts, users should consider hosting these artifacts in their own web-server and configuring Brooklyn to use this. See the documentation for `org.apache.brooklyn.core.entity.drivers.downloads.DownloadProducerFromProperties`.

Troubleshooting

- [Overview](#)
- [Web Console Issues](#)
- [Deployment](#)
- [Server Connectivity](#)
- [Brooklyn Slow or Unresponsive](#)
- [Increase Entropy](#)
- [Increase System Resource Limits](#)
- [Detailed Support Report](#)
- [SoftwareProcess Entities](#)
- [Going Deep in Java and Logs](#)
- [Monitoring Memory Usage](#)

Overview

This guide describes sources of information for understanding when things go wrong.

Whether you're customizing out-of-the-box blueprints, or developing your own custom blueprints, you will inevitably have to deal with `entity` failure. Thankfully Brooklyn provides plenty of information to help you locate and resolve any issues you may encounter.

Web-console Runtime Error Information

Entity Hierarchy

The Brooklyn web-console includes a tree view of the entities within an application. Errors within the application are represented visually, showing a "fire" image on the `entity`.

When an error causes an entire application to be unexpectedly down, the error is generally propagated to the top-level `entity` - i.e. marking it as "on fire". To find the underlying error, one should expand the `entity` hierarchy tree to find the specific entities that have actually failed.

Entity's Error Status

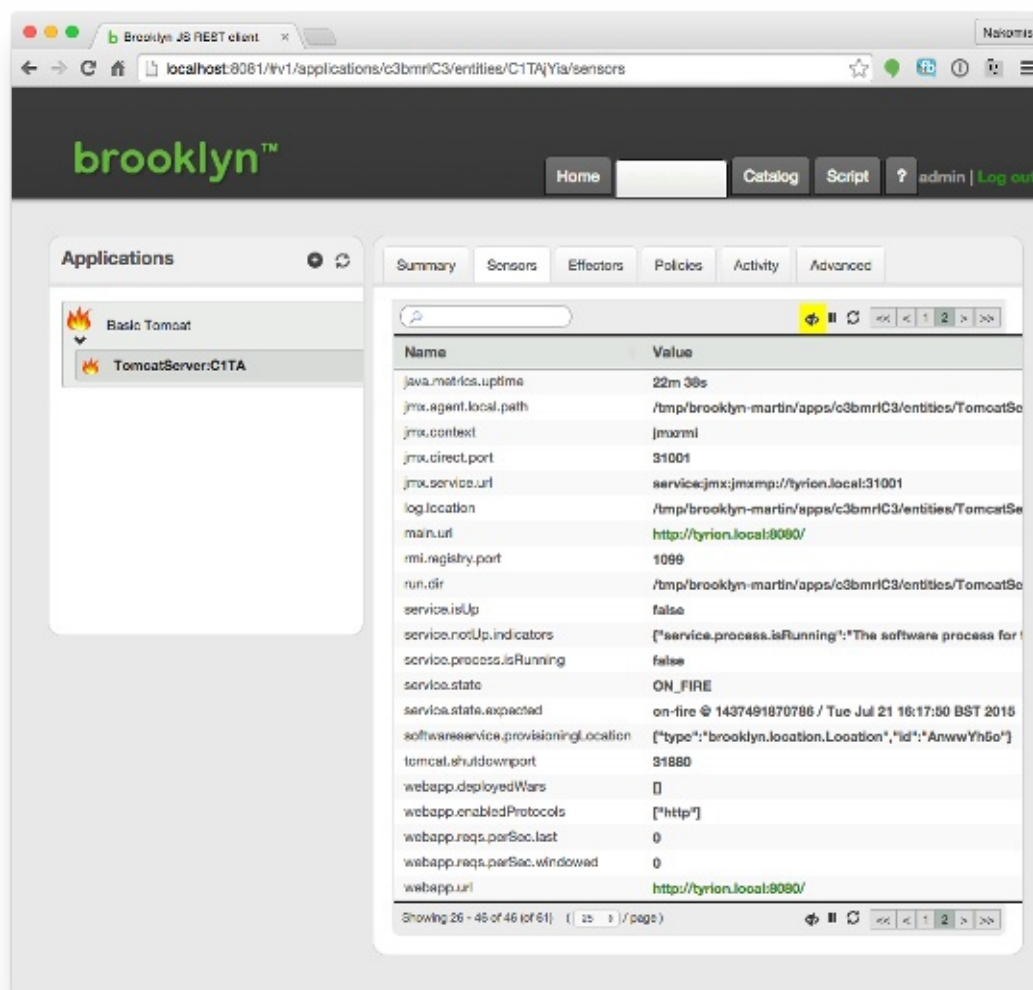
Many entities have some common sensors (i.e. attributes) that give details of the error status:

- `service.isUp` (often referred to as "service up") is a boolean, saying whether the service is up. For many software processes, this is inferred from whether the "service.notUp.indicators" is empty. It is also possible for some entities to set this attribute directly.
- `service.notUp.indicators` is a map of errors. This often gives much more information than the single `service.isUp` attribute. For example, there may be many health-check indicators for a component: is the root URL reachable, is the management api reporting healthy, is the process running, etc.
- `service.problems` is a map of namespaced indicators of problems with a service.
- `service.state` is the actual state of the service - e.g. CREATED, STARTING, RUNNING, STOPPING, STOPPED, DESTROYED and ON_FIRE.
- `service.state.expected` indicates the state the service is expected to be in (and when it transitioned to that). For example, is the service expected to be starting, running, stopping, etc.

These `sensor` values are shown in the "sensors" tab - see below.

Sensors View

The "Sensors" tab in the Brooklyn web-console shows the attribute values of a particular `entity`. This gives lots of runtime information, including about the health of the `entity` - the set of attributes will vary between different `entity` types.



Note that null (or not set) sensors are hidden by default. You can click on the `show/hide empty records` icon (highlighted in yellow above) to see these sensors as well.

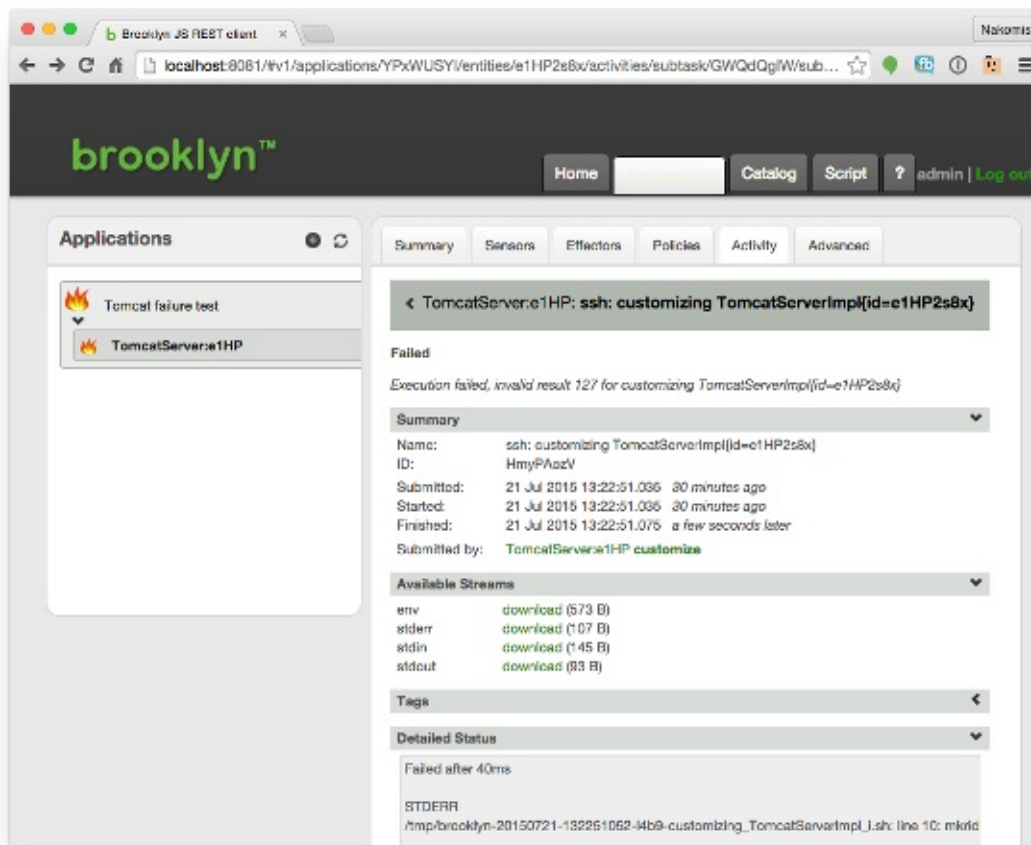
The sensors view is also tabulated. You can configure the numbers of sensors shown per page (at the bottom). There is also a search bar (at the top) to filter the sensors shown.

Activity View

The activity view shows the tasks executed by a given `entity`. The top-level tasks are the effectors (i.e. operations) invoked on that `entity`. This view allows one to drill into the task, to see details of errors.

Select the `entity`, and then click on the `Activities` tab.

In the table showing the tasks, each row is a link - clicking on the row will drill into the details of that task, including sub-tasks:



For ssh tasks, this allows one to drill down to see the env, stdin, stdout and stderr. That is, you can see the commands executed (stdin) and environment variables (env), and the output from executing that (stdout and stderr).

For tasks that did not fail, one can still drill into the tasks to see what was done.

It's always worth looking at the Detailed Status section as sometimes that will give you the information you need. For example, it can show the exception stack trace in the thread that was executing the task that failed.

Log Files

Brooklyn's logging is configurable, for the files created, the logging levels, etc. See [Logging docs](#).

With out-of-the-box logging, `brooklyn.info.log` and `brooklyn.debug.log` files are created. These are by default rolling log files: when the log reaches a given size, it is compressed and a new log file is started. Therefore check the timestamps of the log files to ensure you are looking in the correct file for the time of your error.

With out-of-the-box logging, info, warnings and errors are written to the `brooklyn.info.log` file. This gives a summary of the important actions and errors. However, it does not contain full stacktraces for errors.

To find the exception, we'll need to look in Brooklyn's debug log file. By default, the debug log file is named `brooklyn.debug.log`. You can use your favourite tools for viewing large text files.

One possible tool is `less`, e.g. `less brooklyn.debug.log`. We can quickly find the last exception by navigating to the end of the log file (using `shift-G`), then performing a reverse-lookup by typing `?Exception` and pressing `Enter`. Sometimes an error results in multiple exceptions being logged (e.g. first for the `entity`, then for the cluster, then for the

app). If you know the text of the error message (e.g. copy-pasted from the Activities view of the web-console) then one can search explicitly for that text.

The `grep` command is also extremely helpful. Useful things to grep for include:

- The `entity` id (see the "summary" tab of the `entity` in the web-console for the id).
- The `entity` type name (if there are only a small number of entities of that type).
- The VM IP address.
- A particular error message (e.g. copy-pasted from the Activities view of the web-console).
- The word WARN etc, such as `grep -E "WARN|ERROR" brooklyn.info.log`.

Grep'ing for particular log messages is also useful. Some examples are shown below:

- INFO: "Started application", "Stopping application" and "Stopped application"
- INFO: "Creating VM "
- DEBUG: "Finished VM "

Web Console Issues

Page Does Not Load in Chrome, Saying ""Waiting for available socket..."

If you find that the Web Console does not load in Chrome (giving a message "Waiting for available socket..."), there are two possible explanations.

The first reason is that another tab for the same host:port has a login dialog that is prompting for a username and password. This will block other tabs that are also trying to connect. The solution is to login at the first tab, or to close that tab.

A second possible reason is that there are too many open connections in Chrome to that domain. There is a limit in Chrome for the number of open socket connections to a given domain. If this is exceeded, subsequent tabs that try to connect will wait for an available socket.

For more information, see <http://stackoverflow.com/questions/23679968/chrome-hangs-after-certain-amount-of-data-transferred-waiting-for-available-soc>.

<chrome://net-internals/#sockets> is also a useful diagnostic tool.

Deployment

This guide describes common problems encountered when deploying applications.

YAML deployment errors

The error `Invalid YAML: Plan not in acceptable format: Cannot convert ...` means that the text is not valid **YAML**. Common reasons include that the indentation is incorrect, or that there are non-matching brackets.

The error `Unrecognized application blueprint format: no services defined` means that the `services:` section is missing.

An error like the one shown below means that the given **entity** type (in this case `com.acme.Foo`) is not in the catalog or on the classpath:

```
Deployment plan item Service[name=<null>,description=<null>,serviceType=com.acme.Foo,characteristics=[],customAttributes={}] cannot be matched
```

An error like the one shown below means that the given **location** (in this case `aws-ec3`) was unknown:

```
Illegal parameter for 'location' (aws-ec3); not resolvable: java.util.NoSuchElementException: Unknown location 'aws-ec3': either this location is not recognised or there is a problem with location resolver configuration
```

This means it does not match any of the named locations in `brooklyn.properties`, nor any of the clouds enabled in the jclouds support, nor any of the locations added dynamically through the catalog API.

VM Provisioning Failures

There are many stages at which VM provisioning can fail! An error `Failure running task provisioning` means there was some problem obtaining or connecting to the machine.

An error like `... Not authorized to access cloud ...` usually means the wrong identity/credential was used.

AWS requires a `X-Amz-Date` header which contains the date of the Apache Brooklyn AWS client. If the date on the server is wrong, for example several minutes behind you will get an Authorization Exception. This is to prevent replay attacks. Please be sure to set the clock correctly on the machine running Apache Brooklyn. To set the time on Linux you can use the `ntp` client (e.g. `sudo ntpdate pool.ntp.org`). We advise running the **ntp daemon** so that the clock is kept continually in sync.

An error like `Unable to match required VM template constraints` means that a matching image (e.g. AML in AWS terminology) could not be found. This could be because an incorrect explicit image id was supplied, or because the match-criteria could not be satisfied using the given images available in the given cloud. The first time this error is encountered, a listing of all images in that cloud/region will be written to the debug log.

Failure to form an ssh connection to the newly provisioned VM can be reported in several different ways, depending on the nature of the error. This breaks down into failures at different points:

- Failure to reach the ssh port (e.g. `... could not connect to any ip address port 22 on node ...`).
- Failure to do the very initial ssh login (e.g. `... Exhausted available authentication methods ...`).
- Failure to ssh using the newly created user.

There are many possible reasons for this ssh failure, which include:

- The VM was "dead on arrival" (DOA) - sometimes a cloud will return an unusable VM. One can work around this using the `machineCreateAttempts` configuration option, to automatically retry with a new VM.
- Local network restrictions. On some guest wifis, external access to port 22 is forbidden. Check by manually trying to reach port 22 on a different machine that you have access it.
- NAT rules not set up correctly. On some clouds that have only private IPs, Brooklyn can automatically create NAT rules to provide access to port 22. If this NAT rule creation fails for some reason, then Brooklyn will not be able to reach the VM. If NAT rules are being created for your cloud, then check the logs for warnings or errors about the NAT rule creation.
- ssh credentials incorrectly configured. The Brooklyn configuration is very flexible in how ssh credentials can be configured. However, if a more advanced configuration is used incorrectly (e.g. the wrong login user, or invalid ssh keys) then this will fail.
- Wrong login user. The initial login user to use when first logging into the new VM is inferred from the metadata provided by the cloud provider about that image. This can sometimes be incomplete, so the wrong user may be used. This can be explicitly set using the `loginUser` configuration option. An example of this is with some Ubuntu VMs, where the "ubuntu" user should be used. However, on some clouds it defaults to trying to ssh as "root".
- Bad choice of user. By default, Brooklyn will create a user with the same name as the user running the Brooklyn process; the choice of user name is configurable. If this user already exists on the machine, then the user setup will not behave as expected. Subsequent attempts to ssh using this user could then fail.
- Custom credentials on the VM. Most clouds will automatically set the ssh login details (e.g. in AWS using the key-pair, or in CloudStack by auto-generating a password). However, with some custom images the VM will have hard-coded credentials that must be used. If Brooklyn's configuration does not match that, then it will fail.
- Guest customisation by the cloud. On some clouds (e.g. vCloud Air), the VM can be configured to do guest customisation immediately after the VM starts. This can include changing the root password. If Brooklyn is not configured with the expected changed password, then the VM provisioning may fail (depending if Brooklyn connects before or after the password is changed!).

A very useful debug configuration is to set `destroyOnFailure` to false. This will allow ssh failures to be more easily investigated.

java.security.KeyException when Provisioning VM

The exception `java.security.KeyException` can be thrown when jclouds is attempting the SSL handshake, to make cloud API calls. This can happen if the version of nss is older than 3.16 - the nss package includes the ssl library.

To fix this on CentOS, run:

```
sudo yum upgrade nss
```

For a discussion of investigating this kind of issue, see this [Backslasher blog](#).

The full stacktrace is shown below:

```
Caused by: javax.net.ssl.SSLException: java.security.ProviderException: java.security.KeyException
    at sun.security.ssl.Alerts.getSSLException(Alerts.java:208)
    at sun.security.ssl.SSLSocketImpl.fatal(SSLSocketImpl.java:1949)
    at sun.security.ssl.SSLSocketImpl.fatal(SSLSocketImpl.java:1906)
    at sun.security.ssl.SSLSocketImpl.handleException(SSLSocketImpl.java:1889)
    at sun.security.ssl.SSLSocketImpl.startHandshake(SSLSocketImpl.java:1410)
    at sun.security.ssl.SSLSocketImpl.startHandshake(SSLSocketImpl.java:1387)
    at sun.net.www.protocol.https.HttpsClient.afterConnect(HttpsClient.java:559)
    at sun.net.www.protocol.https.AbstractDelegateHttpsURLConnection.connect(AbstractDelegateHttpsURLConnection
.java:185)
    at sun.net.www.protocol.http.HttpURLConnection.getOutputStream0(HttpURLConnection.java:1283)
```

```

at sun.net.www.protocol.http.HttpURLConnection.getOutputStream(HttpURLConnection.java:1258)
at sun.net.www.protocol.https.HttpsURLConnectionImpl.getOutputStream(HttpsURLConnectionImpl.java:250)
at org.jclouds.http.internal.JavaUrlHttpCommandExecutorService.writePayloadToConnection(JavaUrlHttpCommandE
xecutorService.java:294)
at org.jclouds.http.internal.JavaUrlHttpCommandExecutorService.convert(JavaUrlHttpCommandExecutorService.ja
va:170)
at org.jclouds.http.internal.JavaUrlHttpCommandExecutorService.convert(JavaUrlHttpCommandExecutorService.ja
va:64)
at org.jclouds.http.internal.BaseHttpCommandExecutorService.invoke(BaseHttpCommandExecutorService.java:95)
... 64 more
Caused by: java.security.ProviderException: java.security.KeyException
at sun.security.ec.ECKeyPairGenerator.generateKeyPair(ECKeyPairGenerator.java:147)
at java.security.KeyPairGenerator$Delegate.generateKeyPair(KeyPairGenerator.java:703)
at sun.security.ssl.ECDHCRYPTO.<init>(ECDHCRYPTO.java:77)
at sun.security.ssl.ClientHandshaker.serverKeyExchange(ClientHandshaker.java:721)
at sun.security.ssl.ClientHandshaker.processMessage(ClientHandshaker.java:281)
at sun.security.ssl.Handshaker.processLoop(Handshaker.java:979)
at sun.security.ssl.Handshaker.process_record(Handshaker.java:914)
at sun.security.ssl.SSLSocketImpl.readRecord(SSLSocketImpl.java:1062)
at sun.security.ssl.SSLSocketImpl.performInitialHandshake(SSLSocketImpl.java:1375)
at sun.security.ssl.SSLSocketImpl.startHandshake(SSLSocketImpl.java:1403)
... 74 more
Caused by: java.security.KeyException
at sun.security.ec.ECKeyPairGenerator.generateECKeyPair(Native Method)
at sun.security.ec.ECKeyPairGenerator.generateKeyPair(ECKeyPairGenerator.java:128)
... 83 more

```

Timeout Waiting For Service-Up

A common generic error message is that there was a timeout waiting for service-up.

This just means that the [entity](#) did not get to service-up in the pre-defined time period (the default is two minutes, and can be configured using the `start.timeout` config key; the timer begins after the start tasks are completed).

See the [overview](#) for where to find additional information, especially the section on "[Entity's Error Status](#)".

Invalid packet error

If you receive an error message similar to the one below when provisioning a VM, it means that the wrong username is being used for ssh'ing to the machine. The "invalid packet" is because a response such as "Please login as the ubuntu user rather than root user." is being sent back.

You can workaround the issue by explicitly setting the user that Brooklyn should use to login to the VM (typically the OS default user).

```

error acquiring SFTPClient() (out of retries - max 50)
Invalid packet: indicated length too large
java.lang.IllegalStateException
Invalid packet: indicated length too large

```

An example of how to explicitly set the user is shown below (when defining a [Location](#)) by using 'loginUser':

```

brooklyn.locations:
- type: jclouds:aws-ec2
  brooklyn.config:
    displayName: aws-us-east-1
    region: us-east-1
    identity: <add>
    credential: <add>

```

```
loginUser: centos
```

SSLException close_notify Exception

The following error, when deploying a [blueprint](#), has been shown to be caused by issues with DNS provided by your ISP or traffic filtering such as child-safe type filtering:

```
Caused by: javax.net.ssl.SSLException: Received fatal alert: close_notify
```

To resolve this try disabling traffic filtering and setting your DNS to a public server such as 8.8.8.8 to use google [DNS](#). [See here](#) for details on how to configure this.

Server Connectivity

A common problem when setting up an application in the cloud is getting the basic connectivity right - how do I get my service (e.g. a TCP host:port) publicly accessible over the internet?

This varies a lot - e.g. Is the VM public or in a private network? Is the service only accessible through a load balancer? Should the service be globally reachable or only to a particular CIDR?

This guide gives some general tips for debugging connectivity issues, which are applicable to a range of different service types. Choose those that are appropriate for your use-case.

VM reachable

If the VM is supposed to be accessible directly (e.g. from the public internet, or if in a private network then from a jump host)...

ping

Can you `ping` the VM from the machine you are trying to reach it from?

However, ping is over ICMP. If the VM is unreachable, it could be that the firewall forbids ICMP but still lets TCP traffic through.

telnet to TCP port

You can check if a given TCP port is reachable and listening using `telnet <host> <port>`, such as `telnet www.google.com 80`, which gives output like:

```
Trying 31.55.163.219...
Connected to www.google.com.
Escape character is '^J'.
```

If this is very slow to respond, it can be caused by a firewall blocking access. If it is fast, it could be that the server is just not listening on that port.

DNS and routing

If using a hostname rather than IP, then is it resolving to a sensible IP?

Is the route to the server sensible? (e.g. one can hit problems with proxy servers in a corporate network, or ISPs returning a default result for unknown hosts).

The following commands can be useful:

- `host` is a DNS lookup utility. e.g. `host www.google.com`.
- `dig` stands for "domain information groper". e.g. `dig www.google.com`.
- `traceroute` prints the route that packets take to a network host. e.g. `traceroute www.google.com`.

Proxy settings

Depending on the type of [location](#), brooklyn might use HTTP to provision machines (clocker, jclouds). If the host environment defines proxy settings, these might interfere with the reachability of the respective HTTP service.

One such case is using VirtualBox with host-only or private internal network settings, while using an external proxy for accessing the internet. It is clear that the external proxy won't be able to route HTTP calls properly, but that might not be clear when reading the logs (although brooklyn will present the failing URL).

Try accessing the web-service URLs from a browser via the proxy, or perhaps try running brooklyn with proxy disabled:

```
export http_proxy=
bin/brooklyn launch
```

If a system-level proxy server has been configured, you can instruct brooklyn to use the proxy server by passing `-Djava.net.useSystemProxies=true` to the JVM

Service is listening

Service responds

Try connecting to the service from the VM itself. For example, `curl http://localhost:8080` for a web-service.

On dev/test VMs, don't be afraid to install the utilities you need such as `curl`, `telnet`, `nc`, etc. Cloud VMs often have a very cut-down set of packages installed. For example, execute `sudo apt-get update; sudo apt-get install -y curl` or `sudo yum install -y curl`.

Listening on port

Check that the service is listening on the port, and on the correct NIC(s).

Execute `netstat -antp` (or on OS X `netstat -antp TCP`) to list the TCP ports in use (or use `-anup` for UDP). You should expect to see the something like the output below for a service.

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State	PID/Program name
tcp	0	0	:::8080	:::*	LISTEN	8276/java

In this case a Java process with pid 8276 is listening on port 8080. The local address `:::8080` format means all NICs (in IPv6 address format). You may also see `0.0.0.0:8080` for IPv4 format. If it says `127.0.0.1:8080` then your service will most likely not be reachable externally.

Use `ip addr show` (or the obsolete `ifconfig -a`) to see the network interfaces on your server.

For `netstat`, run with `sudo` to see the pid for all listed ports.

Firewalls

On Linux, check if `iptables` is preventing the remote connection. On Windows, check the Windows Firewall.

If it is acceptable (e.g. it is not a server in production), try turning off the firewall temporarily, and testing connectivity again. Remember to re-enable it afterwards! On CentOS, this is `sudo service iptables stop`. On Ubuntu, use `sudo ufw disable`. On Windows, press the Windows key and type 'Windows Firewall with Advanced Security' to open the firewall tools, then click 'Windows Firewall Properties' and set the firewall state to 'Off' in the Domain, Public and Private profiles.

If you cannot temporarily turn off the firewall, then look carefully at the firewall settings. For example, execute `sudo iptables -n --list` and `iptables -t nat -n --list`.

Cloud firewalls

Some clouds offer a firewall service, where ports need to be explicitly listed to be reachable.

For example, [security groups for EC2-classic] (<http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-network-security.html#ec2-classic-security-groups>) have rules for the protocols and ports to be reachable from specific CIDRs.

Check these settings via the cloud provider's web-console (or API).

Quick test of a listener port

It can be useful to start listening on a given port, and to then check if that port is reachable. This is useful for testing basic connectivity when your service is not yet running, or to a different port to compare behaviour, or to compare with another VM in the network.

The `nc` netcat tool is useful for this. For example, `nc -l 0.0.0.0 8080` will listen on port TCP 8080 on all network interfaces. On another server, you can then run `echo hello from client | nc <hostname> 8080`. If all works well, this will send "hello from client" over the TCP port 8080, which will be written out by the `nc -l` process before exiting.

Similarly for UDP, you use `-lu`.

You may first have to install `nc`, e.g. with `sudo yum install -y nc` Or `sudo apt-get install netcat`.

Cloud load balancers

For some use-cases, it is good practice to use the load balancer service offered by the cloud provider (e.g. [ELB in AWS](#) or the [Cloudstack Load Balancer] (http://docs.cloudstack.apache.org/projects/cloudstack-installation/en/latest/network_setup.html#management-server-load-balancing))

The VMs can all be isolated within a private network, with access only through the load balancer service.

Debugging techniques here include ensuring connectivity from another jump server within the private network, and careful checking of the load-balancer configuration from the Cloud Provider's web-console.

DNAT

Use of DNAT is appropriate for some use-cases, where a particular port on a particular VM is to be made available.

Debugging connectivity issues here is similar to the steps for a cloud load balancer. Ensure connectivity from another jump server within the private network. Carefully check the NAT rules from the Cloud Provider's web-console.

Guest wifi

It is common for guest wifi to restrict access to only specific ports (e.g. 80 and 443, restricting ssh over port 22 etc).

Normally your best bet is then to abandon the guest wifi (e.g. to tether to a mobile phone instead).

There are some unconventional workarounds such as [configuring sshd to listen on port 80 so you can use an ssh tunnel](#). However, the firewall may well inspect traffic so sending non-http traffic over port 80 may still fail.

Brooklyn Slow or Unresponsive

There are many possible causes for a Brooklyn server becoming slow or unresponsive. This guide describes some possible reasons, and some commands and tools that can help diagnose the problem.

Possible reasons include:

- CPU is max'ed out
- Memory usage is extremely high
- SSH'ing is very slow due (e.g. due to lack of entropy)
- Out of disk space

See [Brooklyn Requirements](#) for details of server requirements.

Machine Diagnostics

The following commands will collect OS-level diagnostics about the machine, and about the Brooklyn process. The commands below assume use of CentOS 6.x. Minor adjustments may be required for other platforms.

OS and Machine Details

To display system information, run:

```
uname -a
```

To show details of the CPU and memory available to the machine, run:

```
cat /proc/cpuinfo
cat /proc/meminfo
```

User Limits

To display information about user limits, run the command below (while logged in as the same user who runs Brooklyn):

```
ulimit -a
```

If Brooklyn is run as a different user (e.g. with user name "adalovelace"), then instead run:

```
ulimit -a -u adalovelace
```

Of particular interest is the limit for "open files".

See [Increase System Resource Limits](#) for more information.

Disk Space

The command below will list the disk size for each partition, including the amount used and available. If the Brooklyn base directory, persistence directory or logging directory are close to 0% available, this can cause serious problems:

```
df -h
```

CPU and Memory Usage

To view the CPU and memory usage of all processes, and of the machine as a whole, one can use the `top` command. This runs interactively, updating every few seconds. To collect the output once (e.g. to share diagnostic information in a bug report), run:

```
top -n 1 -b > top.txt
```

File and Network Usage

To count the number of open files for the Brooklyn process (which includes open socket connections):

```
BROOKLYN_HOME=/home/users/brooklyn/apache-brooklyn-0.9.0-bin  
BROOKLYN_PID=$(cat $BROOKLYN_HOME/pid_java)  
lsof -p $BROOKLYN_PID | wc -l
```

To count (or view the number of "established" internet connections, run:

```
netstat -an | grep ESTABLISHED | wc -l
```

Linux Kernel Entropy

A lack of entropy can cause random number generation to be extremely slow. This can cause tasks like ssh to also be extremely slow. See [linux kernel entropy](#) for details of how to work around this.

Process Diagnostics

Thread and Memory Usage

To get memory and thread usage for the Brooklyn (Java) process, two useful tools are `jstack` and `jmap`. These require the "development kit" to also be installed (e.g. `yum install java-1.8.0-openjdk-devel`). Some useful commands are shown below:

```
BROOKLYN_HOME=/home/users/brooklyn/apache-brooklyn-0.9.0-bin  
BROOKLYN_PID=$(cat $BROOKLYN_HOME/pid_java)  
  
jstack $BROOKLYN_PID  
jmap -histo:live $BROOKLYN_PID  
jmap -heap $BROOKLYN_PID
```

Runnable Threads

The `jstack-active` script is a convenient light-weight way to quickly see which threads of a running Brooklyn server are attempting to consume the CPU. It filters the output of `jstack`, to show only the "really-runnable" threads (as opposed to those that are blocked).

```
BROOKLYN_HOME=/home/users/brooklyn/apache-brooklyn-0.9.0-bin
```

```
BROOKLYN_PID=$(cat $BROOKLYN_HOME/pid_java)

curl -O https://raw.githubusercontent.com/apache/brooklyn-dist/master/scripts/jstack-active.sh

jstack-active $BROOKLYN_PID
```

Profiling

If an in-depth investigation of the CPU usage (and/or object creation) of a Brooklyn Server is requiring, there are many profiling tools designed specifically for this purpose. These generally require that the process be launched in such a way that a profiler can attach, which may not be appropriate for a production server.

Remote Debugging

If the Brooklyn Server was originally run to allow a remote debugger to connect (strongly discouraged in production!), then this provides a convenient way to investigate why Brooklyn is being slow or unresponsive. See the Debugging Tips in the tip [Debugging Remote Brooklyn](#) and the [IDE docs](#) for more information.

Log Files

Apache Brooklyn will by default create `brooklyn.info.log` and `brooklyn.debug.log` files. See the [Logging](#) docs for more information.

The following are useful log messages to search for (e.g. using `grep`). Note the wording of these messages (or their very presence) may change in future version of Brooklyn.

Normal Logging

The lines below are commonly logged, and can be useful to search for when finding the start of a section of logging.

```
2016-05-30 17:05:51,458 INFO  o.a.b.l.BrooklynWebServer [main]: Started Brooklyn console at http://127.0.0.1:8081/, running classpath://brooklyn.war
2016-05-30 17:06:04,098 INFO  o.a.b.c.m.h.HighAvailabilityManagerImpl [main]: Management node tF3GPvQ5 running as HA MASTER autodetected
2016-05-30 17:06:08,982 INFO  o.a.b.c.m.r.InitialFullRebindIteration [brooklyn-execmanager-rvpnFTeL-0]: Rebinding from /home/compose/compose-amp-state/brooklyn-persisted-state/data for master rvpnFTeL...
2016-05-30 17:06:11,105 INFO  o.a.b.c.m.r.RebindIteration [brooklyn-execmanager-rvpnFTeL-0]: Rebind complete (MASTER) in 2s: 19 apps, 54 entities, 50 locations, 46 policies, 704 enrichers, 0 feeds, 160 catalog items
```

Memory Usage

The debug log includes (every minute) a log statement about the memory usage and task activity. For example:

```
2016-05-27 12:20:19,395 DEBUG o.a.b.c.m.i.BrooklynGarbageCollector [brooklyn-gc]: brooklyn gc (before) - using 328 MB / 496 MB memory (5.58 kB soft); 69 threads; storage: {datagrid={size=7, createCount=7}, refsMapSize=0, listsMapSize=0}; tasks: 10 active, 33 unfinished; 78 remembered, 1696906 total submitted)
2016-05-27 12:20:19,395 DEBUG o.a.b.c.m.i.BrooklynGarbageCollector [brooklyn-gc]: brooklyn gc (after) - using 328 MB / 496 MB memory (5.58 kB soft); 69 threads; storage: {datagrid={size=7, createCount=7}, refsMapSize=0, listsMapSize=0}; tasks: 10 active, 33 unfinished; 78 remembered, 1696906 total submitted)
```

These can be extremely useful if investigating a memory or thread leak, or to determine whether a surprisingly high number of tasks are being executed.

Subscriptions

One source of high CPU in Brooklyn is when a subscription (e.g. for a [policy](#) or [enricher](#)) is being triggered many times (i.e. handling many events). A log message like that below will be logged on every 1000 events handled by a given single subscription.

```
2016-05-30 17:29:09,125 DEBUG o.a.b.c.m.i.LocalSubscriptionManager [brooklyn-execmanager-rvpnFTeL-8]: 1000 events for subscriber Subscription[SCFnav9g;CanopyComposeApp{id=gIeTwhU2}@gIeTwhU2:webapp.url]
```

If a subscription is handling a huge number of events, there are a couple of common reasons:

- first, it could be subscribing to too much activity - e.g. a wildcard subscription for all events from all entities.
- second it could be an infinite loop (e.g. where an [enricher](#) responds to a [sensor](#)-changed event by setting that same [sensor](#), thus triggering another [sensor](#)-changed event).

User Activity

All activity triggered by the REST API or web-console will be logged. Some examples are shown below:

```
2016-05-19 17:52:30,150 INFO o.a.b.r.r.ApplicationResource [brooklyn-jetty-server-8081-qtp1058726153-17473]: Launched from YAML: name: My Example App
location: aws-ec2:us-east-1
services:
- type: org.apache.brooklyn.entity.webapp.tomcat.TomcatServer

2016-05-30 14:46:19,516 DEBUG brooklyn.REST [brooklyn-jetty-server-8081-qtp1104967201-20881]: Request Tisj14 starting: POST /v1/applications/NiBy0v8Q/entities/NiBy0v8Q/expunge from 77.70.102.66
```

Entity Activity

If investigating the behaviour of a particular [entity](#) (e.g. on failure), it can be very useful to `grep` the info and debug log for the [entity](#)'s id. For a software process, the debug log will include the stdout and stderr of all the commands executed by that [entity](#).

It can also be very useful to search for all [effector](#) invocations, to see where the behaviour has been triggered:

```
2016-05-27 12:45:43,529 DEBUG o.a.b.c.m.i.EffectorUtils [brooklyn-execmanager-gvP7MuZF-14364]: Invoking effector stop on TomcatServerImpl{id=mPujYmPd}
```

Increase Entropy

Checking entropy level

A lack of entropy can cause random number generation to be extremely slow. This results in tasks like ssh to also be extremely slow. One can check the available entropy on a machine by running the command:

```
cat /proc/sys/kernel/random/entropy_avail
```

It should be a value above 2000.

If you are installing Apache Brooklyn on a virtual machine, you may find that it has insufficient entropy. You may need to increase the Linux kernel entropy in order to speed up the ssh connections to the managed entities. You can install and configure `rng-tools`, or just use `/dev/urandom`.

Installing rng-tool

If you are using a RHEL 6 based OS:

```
sudo -i
yum -y -q install rng-tools
echo "EXTRAOPTIONS=\"-r /dev/urandom\"" | cat >> /etc/sysconfig/rngd
/etc/init.d/rngd start
```

If you are using a RHEL 7 or a systemd based system:

```
sudo yum -y -q install rng-tools

# Configure rng to use /dev/urandom
# Change the "ExecStart" line to:
# ExecStart=/sbin/rngd -f -r /dev/urandom
sudo vi /etc/systemd/system/multi-user.target.wants/rngd.service

sudo systemctl daemon-reload
sudo systemctl start rngd
```

If you are using a Debian-based OS:

```
sudo -i
apt-get -y install rng-tools
echo "HRNGDEVICE=/dev/urandom" | cat >> /etc/default/rng-tools
/etc/init.d/rng-tools start
```

Using /dev/urandom

You can also just `mv /dev/random` then create it again linked to `/dev/urandom` :

```
sudo mv /dev/random /dev/random-real
sudo ln -s /dev/urandom /dev/random
```

Notice! If you map `/dev/random` to use `/dev/urandom` you will need to restart the Apache Brooklyn java process in order for the change to take place.

More Information

The following links contain further information:

- [haveged \(another solution\)](#) and [general info from Digital Ocean](#)
- for specific OSs:
 - [for RHEL or CentOS](#)
 - [for Ubuntu](#)
 - [for Alpine](#)

Increase System Resource Limits

If you encounter the following error:

```
Caused by: java.io.IOException: Too many open files
    at java.io.UnixFileSystem.createFileExclusively(Native Method):1.8.0
```

Please check the limit for opened files `cat /proc/sys/fs/file-max` and increase it. You can increase the maximum limit of opened files by setting `fs.file-max` in `/etc/sysctl.conf` . and then running `sudo sysctl -p` to apply the changes.

If you encounter the error below, e.g. when running with many entities, please consider **increasing the ulimit**:

```
java.lang.OutOfMemoryError: unable to create new native thread
```

On the VM running Apache Brooklyn, it is recommended that `nproc` and `nofile` are reasonably high (e.g. 16384 or higher; a value of 1024 is often the default).

If you want to check the current limits run `ulimit -a` . Alternatively, if Brooklyn is run as a different user (e.g. with user name "brooklyn"), then instead run `ulimit -a -u brooklyn` .

For RHEL (and CentOS) distributions, you can increase the limits by running `sudo vi /etc/security/limits.conf` and adding (if it is "brooklyn" user running Apache Brooklyn):

brooklyn	soft	nproc	16384
brooklyn	hard	nproc	16384
brooklyn	soft	nofile	16384
brooklyn	hard	nofile	16384

Generally you do not have to reboot to apply ulimit values. They are set per session. So after you have the correct values, quit the ssh session and log back in.

For more details, see one of the many posts such as <http://tuxgen.blogspot.co.uk/2014/01/centosrhel-ulimit-and-maximum-number-of.html>.

Detailed Support Report

If you wish to send a detailed report, then depending on the nature of the problem, consider collecting the following information.

See [Brooklyn Slow or Unresponse](#) docs for details of these commands.

```
BROOKLYN_HOME=/home/users/brooklyn/apache-brooklyn-0.9.0-bin
BROOKLYN_PID=$(cat $BROOKLYN_HOME/pid_java)
REPORT_DIR=/tmp/brooklyn-report/
DEBUG_LOG=${BROOKLYN_HOME}/brooklyn.debug.log

uname -a > ${REPORT_DIR}/uname.txt
df -h > ${REPORT_DIR}/df.txt
cat /proc/cpuinfo > ${REPORT_DIR}/cpuinfo.txt
cat /proc/meminfo > ${REPORT_DIR}/meminfo.txt
ulimit -a > ${REPORT_DIR}/ulimit.txt
cat /proc/${BROOKLYN_PID}/limits >> ${REPORT_DIR}/ulimit.txt
top -n 1 -b > ${REPORT_DIR}/top.txt
lsof -p ${BROOKLYN_PID} > ${REPORT_DIR}/lsof.txt
netstat -an > ${REPORT_DIR}/netstat.txt

jmap -histo:live ${BROOKLYN_PID} > ${REPORT_DIR}/jmap-histo.txt
jmap -heap ${BROOKLYN_PID} > ${REPORT_DIR}/jmap-heap.txt
for i in {1..10}; do
    jstack ${BROOKLYN_PID} > ${REPORT_DIR}/jstack.${i}.txt
    sleep 1
done
grep "brooklyn gc" ${DEBUG_LOG} > ${REPORT_DIR}/brooklyn-gc.txt
grep "events for subscriber" ${DEBUG_LOG} > ${REPORT_DIR}/events-for-subscriber.txt
tar czf brooklyn-report.tgz ${REPORT_DIR}
```

Also consider providing your log files and persisted state, though extreme care should be taken if these might contain cloud or machine credentials (especially if [Externalised Configuration](#) is not being used for credential storage).

SoftwareProcess Entities

The [troubleshooting overview](#) in Brooklyn gives information for how to find more information about errors.

If that doesn't give enough information to diagnose, fix or workaround the problem, then it can be required to login to the machine, to investigate further. This guide applies to entities that are types of "SoftwareProcess" in Brooklyn, or that follows those conventions.

VM connection details

The ssh connection details for an [entity](#) is published to a [sensor](#) `host.sshAddress`. The login credentials will depend on the Brooklyn configuration. The default is to use the `~/.ssh/id_rsa` or `~/.ssh/id_dsa` on the Brooklyn host (uploading the associated `~/.ssh/id_rsa.pub` to the machine's `authorized_keys`). However, this can be overridden (e.g. with specific passwords etc) in the [location](#)'s configuration.

For Windows, there is a similar [sensor](#) with the name `host.winrmAddress`.

Install and Run Directories

For ssh-based software processes, the install directory and the run directory are published as sensors `install.dir` and `run.dir` respectively.

For some entities, files are unpacked into the install dir; configuration files are written to the run dir along with log files. For some other entities, these directories may be mostly empty - e.g. if installing RPMs, and that software writes its logs to a different standard [location](#).

Most entities have a [sensor](#) `log.location`. It is generally worth checking this, along with other files in the run directory (such as console output).

Process and OS Health

It is worth checking that the process is running, e.g. using `ps aux` to look for the desired process. Some entities also write the pid of the process to `pid.txt` in the run directory.

It is also worth checking if the required port is accessible. This is discussed in the guide "Troubleshooting Server Connectivity Issues in the Cloud", including listing the ports in use: execute `netstat -antp` (or on OS X `netstat -antp tcp`) to list the TCP ports in use (or use `-anup` for UDP).

It is also worth checking the disk space on the server, e.g. using `df -m`, to check that there is sufficient space on each of the required partitions.

Going Deep in Java and Logs

This guide takes a deep look at the Java and log messages for some failure scenarios, giving common steps used to identify the issues.

Script Failure

Many blueprints run bash scripts as part of the installation. This section highlights how to identify a problem with a bash script.

First let's take a look at the `customize()` method of the Tomcat server [blueprint](#):

```
@Override
public void customize() {
    newScript(CUSTOMIZING)
        .body.append("mkdir -p conf logs webapps temp")
        .failOnNonZeroResultCode()
        .execute();

    copyTemplate(entity.getConfig(TomcatServer.SERVER_XML_RESOURCE), Os.mergePaths(getRunDir(), "conf", "server.xml"));
    copyTemplate(entity.getConfig(TomcatServer.WEB_XML_RESOURCE), Os.mergePaths(getRunDir(), "conf", "web.xml"));

    if (isProtocolEnabled("HTTPS")) {
        String keystoreUrl = Preconditions.checkNotNull(getSslKeystoreUrl(), "keystore URL must be specified if using HTTPS for " + entity);
        String destinationSslKeystoreFile = getHttpsSslKeystoreFile();
        InputStream keystoreStream = resource.getResourceFromUrl(keystoreUrl);
        getMachine().copyTo(keystoreStream, destinationSslKeystoreFile);
    }

    getEntity().deployInitialWars();
}
```

Here we can see that it's running a script to create four directories before continuing with the customization. Let's introduce an error by changing `mkdir` to `mkrid`:

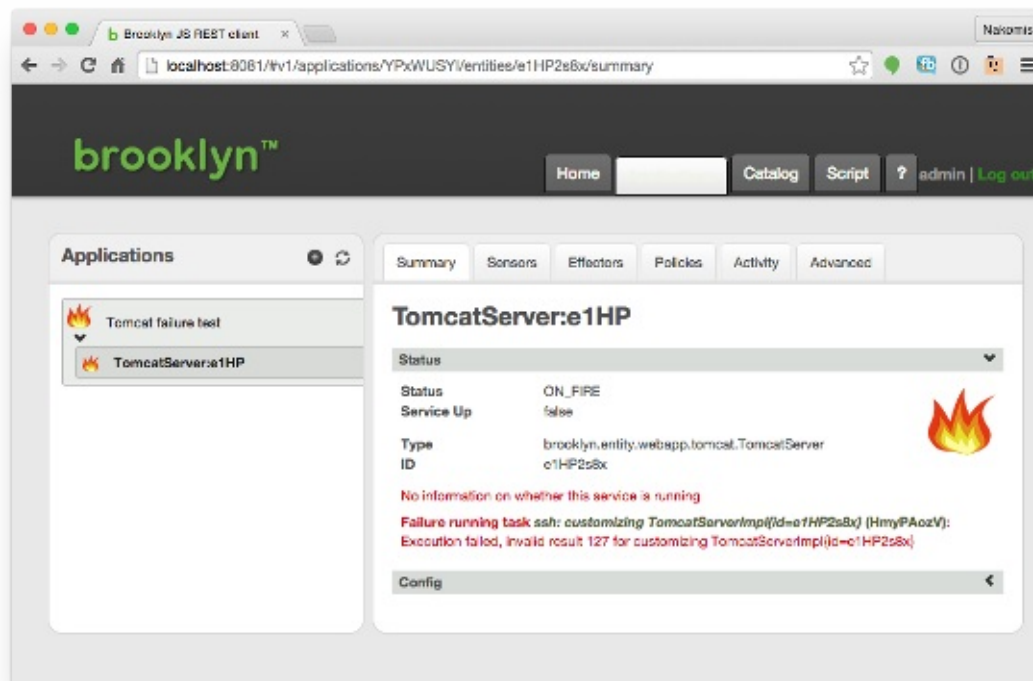
```
newScript(CUSTOMIZING)
    .body.append("mkrid -p conf logs webapps temp") // `mkdir` changed to `mkrid`
    .failOnNonZeroResultCode()
    .execute();
```

Now let's try deploying this using the following [YAML](#):

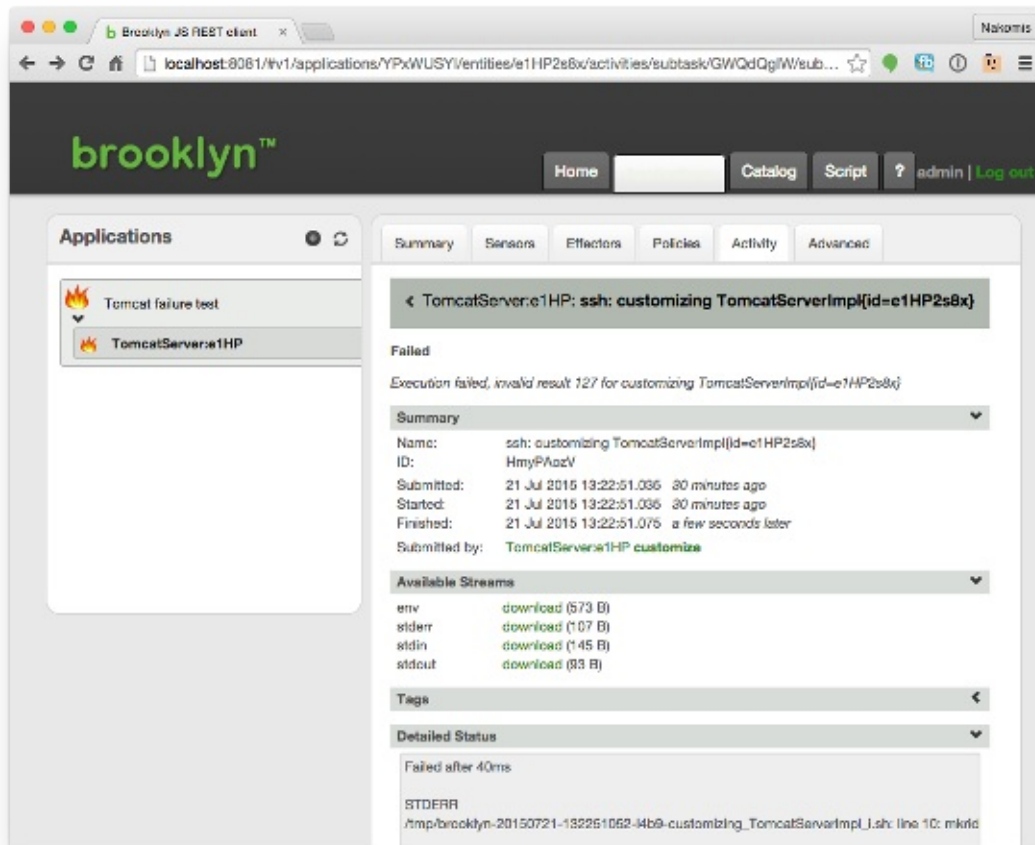
```
name: Tomcat failure test
location: localhost
services:
- type: org.apache.brooklyn.entity.webapp.tomcat.TomcatServer
```

Shortly after deployment, the [entity](#) fails with the following error:

```
Failure running task ssh: customizing TomcatServerImpl{id=e1HP2s8x} (HmyPAozV): Execution failed, invalid result 127 for customizing TomcatServerImpl{id=e1HP2s8x}
```



By selecting the `Activities` tab, we can drill into the task that failed. The list of tasks shown (where the effectors are shown as top-level tasks) are clickable links. Selecting that row will show the details of that particular task, including its sub-tasks. We can eventually get to the specific sub-task that failed:



By clicking on the `stderr` link, we can see the script failed with the following error:

```
/tmp/brooklyn-20150721-132251052-14b9-customizing_TomcatServerImpl_i.sh: line 10: mkrid: command not found
```

This tells us *what* went wrong, but doesn't tell us *where*. In order to find that, we'll need to look at the stack trace that was logged when the exception was thrown.

It's always worth looking at the Detailed Status section as sometimes this will give you the information you need. In this case, the stack trace is limited to the thread that was used to execute the task that ran the script:

```
Failed after 40ms

STDERR
/tmp/brooklyn-20150721-132251052-14b9-customizing_TomcatServerImpl_i.sh: line 10: mkrid: command not found

STDOUT
Executed /tmp/brooklyn-20150721-132251052-14b9-customizing_TomcatServerImpl_i.sh, result 127: Execution failed,
invalid result 127 for customizing TomcatServerImpl{id=e1HP2s8x}

java.lang.IllegalStateException: Execution failed, invalid result 127 for customizing TomcatServerImpl{id=e1HP2
s8x}
    at org.apache.brooklyn.entity.software.base.lifecycle.ScriptHelper.logWithDetailsAndThrow(ScriptHelper.java
:390)
    at org.apache.brooklyn.entity.software.base.lifecycle.ScriptHelper.executeInternal(ScriptHelper.java:379)
    at org.apache.brooklyn.entity.software.base.lifecycle.ScriptHelper$8.call(ScriptHelper.java:289)
    at org.apache.brooklyn.entity.software.base.lifecycle.ScriptHelper$8.call(ScriptHelper.java:287)
    at org.apache.brooklyn.core.util.task.DynamicSequentialTask$DstJob.call(DynamicSequentialTask.java:343)
    at org.apache.brooklyn.core.util.task.BasicExecutionManager$SubmissionCallable.call(BasicExecutionManager.j
```

```
ava:469)
  at java.util.concurrent.FutureTask.run(FutureTask.java:262)
  at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
  at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
  at java.lang.Thread.run(Thread.java:745)
```

In order to find the exception, we'll need to look in Brooklyn's debug log file. By default, the debug log file is named `brooklyn.debug.log`. Usually the easiest way to navigate the log file is to use `less`, e.g. `less brooklyn.debug.log`. We can quickly find the stack trace by first navigating to the end of the log file with `shift-G`, then performing a reverse-lookup by typing `?Tomcat` and pressing `Enter`. If searching for the `blueprint` type (in this case Tomcat) simply matches tasks unrelated to the exception, you can also search for the text of the error message, in this case `?invalid result 127`. You can make the search case-insensitivity by typing `-i` before performing the search. To skip the current match and move to the next one (i.e. 'up' as we're performing a reverse-lookup), simply press `n`.

In this case, the `?Tomcat` search takes us directly to the full stack trace (Only the last part of the trace is shown here):

```
... at com.google.common.util.concurrent.ForwardingFuture.get(ForwardingFuture.java:63) ~[guava-17.0.jar:na]
  at org.apache.brooklyn.core.util.task.BasicTask.get(BasicTask.java:343) ~[classes/:na]
  at org.apache.brooklyn.core.util.task.BasicTask.getUnchecked(BasicTask.java:352) ~[classes/:na]
  ... 9 common frames omitted
Caused by: brooklyn.util.exceptions.PropagatedRuntimeException:
  at org.apache.brooklyn.util.exceptions.Exceptions.propagate(Exceptions.java:97) ~[classes/:na]
  at org.apache.brooklyn.core.util.task.BasicTask.getUnchecked(BasicTask.java:354) ~[classes/:na]
  at org.apache.brooklyn.entity.software.base.lifecycle.ScriptHelper.execute(ScriptHelper.java:339) ~[classes/:na]
  at org.apache.brooklyn.entity.webapp.tomcat.TomcatSshDriver.customize(TomcatSshDriver.java:72) ~[classes/:na]
  at org.apache.brooklyn.entity.software.base.AbstractSoftwareProcessDriver$8.run(AbstractSoftwareProcessDriver.java:150) ~[classes/:na]
  at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471) ~[na:1.7.0_71]
  at org.apache.brooklyn.core.util.task.DynamicSequentialTask$DstJob.call(DynamicSequentialTask.java:343) ~[classes/:na]
  ... 5 common frames omitted
Caused by: java.util.concurrent.ExecutionException: java.lang.IllegalStateException: Execution failed, invalid result 127 for customizing TomcatServerImpl{id=e1HP2s8x}
  at java.util.concurrent.FutureTask.report(FutureTask.java:122) [na:1.7.0_71]
  at java.util.concurrent.FutureTask.get(FutureTask.java:188) [na:1.7.0_71]
  at com.google.common.util.concurrent.ForwardingFuture.get(ForwardingFuture.java:63) ~[guava-17.0.jar:na]
  at org.apache.brooklyn.core.util.task.BasicTask.get(BasicTask.java:343) ~[classes/:na]
  at org.apache.brooklyn.core.util.task.BasicTask.getUnchecked(BasicTask.java:352) ~[classes/:na]
  ... 10 common frames omitted
Caused by: java.lang.IllegalStateException: Execution failed, invalid result 127 for customizing TomcatServerImpl{id=e1HP2s8x}
  at org.apache.brooklyn.entity.software.base.lifecycle.ScriptHelper.logWithDetailsAndThrow(ScriptHelper.java:390) ~[classes/:na]
  at org.apache.brooklyn.entity.software.base.lifecycle.ScriptHelper.executeInternal(ScriptHelper.java:379) ~[classes/:na]
  at org.apache.brooklyn.entity.software.base.lifecycle.ScriptHelper$8.call(ScriptHelper.java:289) ~[classes/:na]
  at org.apache.brooklyn.entity.software.base.lifecycle.ScriptHelper$8.call(ScriptHelper.java:287) ~[classes/:na]
  ... 6 common frames omitted
```

Brooklyn's use of tasks and helper classes can make the stack trace a little harder than usual to follow, but a good place to start is to look through the stack trace for the node's implementation or ssh driver classes (usually named `FooNodeImpl` or `FooSshDriver`). In this case we can see the following:

```
at org.apache.brooklyn.entity.webapp.tomcat.TomcatSshDriver.customize(TomcatSshDriver.java:72) ~[classes/:na]
```

Combining this with the error message of `mkrid: command not found` we can see that indeed `mkdir` has been misspelled `mkrid` on line 72 of `TomcatSshDriver.java`.

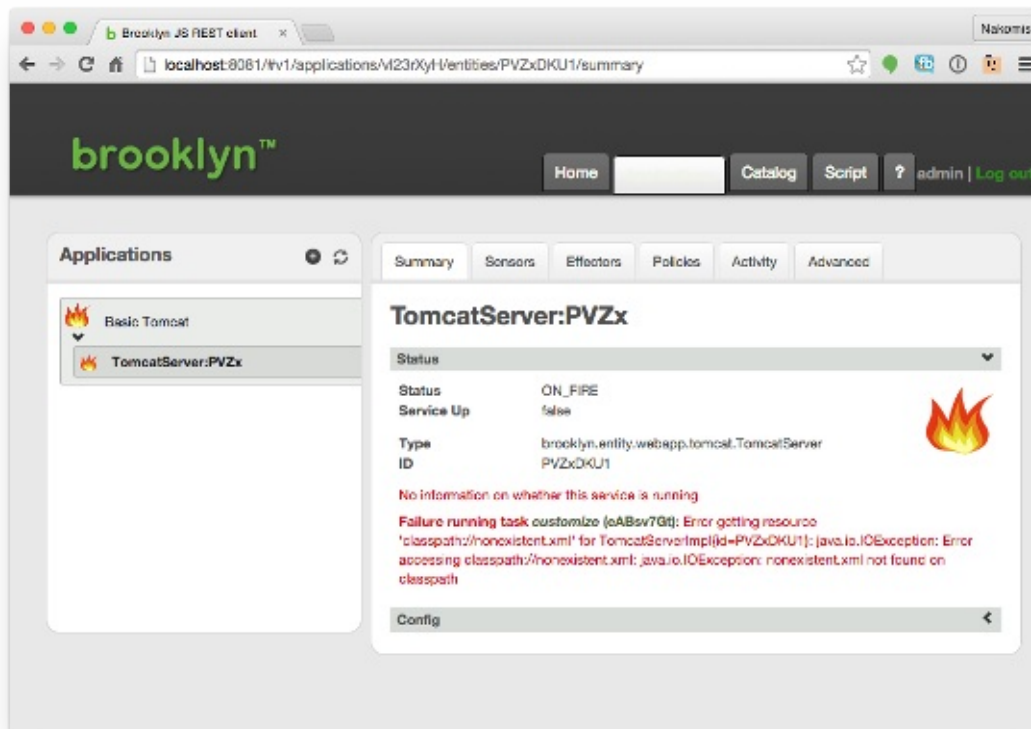
Non-Script Failure

The section above gives an example of a failure that occurs when a script is run. In this section we will look at a failure in a non-script related part of the code. We'll use the `customize()` method of the Tomcat server again, but this time, we'll correct the spelling of 'mkdir' and add a line that attempts to copy a nonexistent resource to the remote server:

```
newScript(CUSTOMIZING)
  .body.append("mkdir -p conf logs webapps temp")
  .failOnNonZeroResultCode()
  .execute();

copyTemplate(entity.getConfig(TomcatServer.SERVER_XML_RESOURCE), Os.mergePaths(getRunDir(), "conf", "server.xml")
));
copyTemplate(entity.getConfig(TomcatServer.WEB_XML_RESOURCE), Os.mergePaths(getRunDir(), "conf", "web.xml"));
copyTemplate("classpath://nonexistent.xml", Os.mergePaths(getRunDir(), "conf", "nonexistent.xml")); // Resource
does not exist!
```

Let's deploy this using the same [YAML](#) from above. Here's the resulting error in the Brooklyn debug console:



Again, this tells us *what* the error is, but we need to find *where* the code is that attempts to copy this file. In this case it's shown in the Detailed Status section, and we don't need to go to the log file:

```
Failed after 221ms: Error getting resource 'classpath://nonexistent.xml' for TomcatServerImpl{id=PVZxDKU1}: java
a.io.IOException: Error accessing classpath://nonexistent.xml: java.io.IOException: nonexistent.xml not found o
n classpath

java.lang.RuntimeException: Error getting resource 'classpath://nonexistent.xml' for TomcatServerImpl{id=PVZxDK
U1}: java.io.IOException: Error accessing classpath://nonexistent.xml: java.io.IOException: nonexistent.xml not
found on classpath
```

```

    at org.apache.brooklyn.core.util.ResourceUtils.getResourceFromUrl(ResourceUtils.java:297)
    at org.apache.brooklyn.core.util.ResourceUtils.getResourceAsString(ResourceUtils.java:475)
    at org.apache.brooklyn.entity.software.base.AbstractSoftwareProcessDriver.getResourceAsString(AbstractSoftwareProcessDriver.java:447)
    at org.apache.brooklyn.entity.software.base.AbstractSoftwareProcessDriver.processTemplate(AbstractSoftwareProcessDriver.java:469)
    at org.apache.brooklyn.entity.software.base.AbstractSoftwareProcessDriver.copyTemplate(AbstractSoftwareProcessDriver.java:390)
    at org.apache.brooklyn.entity.software.base.AbstractSoftwareProcessDriver.copyTemplate(AbstractSoftwareProcessDriver.java:379)
    at org.apache.brooklyn.entity.webapp.tomcat.TomcatSshDriver.customize(TomcatSshDriver.java:79)
    at org.apache.brooklyn.entity.software.base.AbstractSoftwareProcessDriver$8.run(AbstractSoftwareProcessDriver.java:150)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471)
    at org.apache.brooklyn.core.util.task.DynamicSequentialTask$DstJob.call(DynamicSequentialTask.java:343)
    at org.apache.brooklyn.core.util.task.BasicExecutionManager$SubmissionCallable.call(BasicExecutionManager.java:469)
    at java.util.concurrent.FutureTask.run(FutureTask.java:262)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
    at java.lang.Thread.run(Thread.java:745)
Caused by: java.io.IOException: Error accessing classpath://nonexistent.xml: java.io.IOException: nonexistent.xml not found on classpath
    at org.apache.brooklyn.core.util.ResourceUtils.getResourceFromUrl(ResourceUtils.java:233)
    ... 14 more
Caused by: java.io.IOException: nonexistent.xml not found on classpath
    at org.apache.brooklyn.core.util.ResourceUtils.getResourceViaClasspath(ResourceUtils.java:372)
    at org.apache.brooklyn.core.util.ResourceUtils.getResourceFromUrl(ResourceUtils.java:230)
    ... 14 more

```

Looking for `Tomcat` in the stack trace, we can see in this case the problem lies at line 79 of `TomcatSshDriver.java`

External Failure

Sometimes an `entity` will fail outside the direct commands issues by Brooklyn. When installing and launching an `entity`, Brooklyn will check the return code of scripts that were run to ensure that they completed successfully (i.e. the return code of the script is zero). It is possible, for example, that a launch script completes successfully, but the `entity` fails to start.

We can simulate this type of failure by launching Tomcat with an invalid configuration file. As seen in the previous examples, Brooklyn copies two xml configuration files to the server: `server.xml` and `web.xml`

The first few non-comment lines of `server.xml` are as follows (you can see the full file [here](#)):

```

<Server port="${driver.shutdownPort?c}" shutdown="SHUTDOWN">
  <Listener className="org.apache.catalina.core.AprLifecycleListener" SSLEngine="on" />
  <Listener className="org.apache.catalina.core.JasperListener" />

{% endhighlight%}

```

Let's add an unmatched XML element, which will make this XML file invalid:

```

```xml
<Server port="${driver.shutdownPort?c}" shutdown="SHUTDOWN">
 <unmatched-element> <!-- This is invalid XML as we won't add </unmatched-element> -->
 <Listener className="org.apache.catalina.core.AprLifecycleListener" SSLEngine="on" />
 <Listener className="org.apache.catalina.core.JasperListener" />

{% endhighlight%}

```

As Brooklyn doesn't know how these types of resources are used, they're not validated as they're copied to the



```

remote machine.
As far as Brooklyn is concerned, the file will have copied successfully.

Let's deploy Tomcat again, using the same YAML as before. This time, the deployment runs for a few minutes before failing
with `Timeout waiting for SERVICE_UP`:

[![External error in the Brooklyn debug console.](images/external-error.png)](images/external-error-large.png)

If we drill down into the tasks in the `Activities` tab, we can see that all of the installation and launch tasks
completed successfully, and stdout of the `launch` script is as follows:

```console

Executed /tmp/brooklyn-20150721-153049139-fK2U-launching_TomcatServerImpl_id_.sh, result 0

```

The task that failed was the `post-start` task, and the stack trace from the Detailed Status section is as follows:

```

Failed after 5m 1s: Timeout waiting for SERVICE_UP from TomcatServerImpl{id=BUHgQe0s}

java.lang.IllegalStateException: Timeout waiting for SERVICE_UP from TomcatServerImpl{id=BUHgQe0s}
    at org.apache.brooklyn.core.entity.Entities.waitForServiceUp(Entities.java:1073)
    at org.apache.brooklyn.entity.software.base.SoftwareProcessImpl.waitForServiceUp(SoftwareProcessImpl.java:388)
    at org.apache.brooklyn.entity.software.base.SoftwareProcessImpl.waitForServiceUp(SoftwareProcessImpl.java:385)
    at org.apache.brooklyn.entity.software.base.SoftwareProcessDriverLifecycleEffectorTasks.postStartCustom(SoftwareProcessDriverLifecycleEffectorTasks.java:164)
    at org.apache.brooklyn.entity.software.base.lifecycle.MachineLifecycleEffectorTasks$7.run(MachineLifecycleEffectorTasks.java:433)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471)
    at org.apache.brooklyn.core.util.task.DynamicSequentialTask$DstJob.call(DynamicSequentialTask.java:343)
    at org.apache.brooklyn.core.util.task.BasicExecutionManager$SubmissionCallable.call(BasicExecutionManager.java:469)
    at java.util.concurrent.FutureTask.run(FutureTask.java:262)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
    at java.lang.Thread.run(Thread.java:745)

```

This doesn't really tell us what we need to know, and looking in the `brooklyn.debug.log` file yields no further clues. The key here is the error message `Timeout waiting for SERVICE_UP`. After running the installation and launch scripts, assuming all scripts completed successfully, Brooklyn will periodically check the health of the node and will set the node on fire if the health check does not pass within a pre-prescribed period (the default is two minutes, and can be configured using the `start.timeout` config key). The periodic health check also continues after the successful launch in order to check continued operation of the node, but in this case it fails to pass at all.

The first thing we need to do is to find out how Brooklyn determines the health of the node. The health-check is often implemented in the `isRunning()` method in the `entity`'s ssh driver. Tomcat's implementation of `isRunning()` is as follows:

```

@Override
public boolean isRunning() {
    return newScript(ImmutableMap.of(USE_PID_FILE, "pid.txt"), CHECK_RUNNING).execute() == 0;
}

```

The `newScript` method has conveniences for default scripts to check if a process is running based on its PID. In this case, it will look for Tomcat's PID in the `pid.txt` file and check if the PID is the PID of a running process

It's worth a quick sanity check at this point to check if the PID file exists, and if the process is running. By default, the pid file is located in the run directory of the `entity`. You can find the `location` of the `entity`'s run directory by looking at the `run.dir` `sensor`. In this case it is `/tmp/brooklyn-martin/apps/jIzIHxTP/entities/TomcatServer_BUHgQe0s`. To find the pid, you simply cat the `pid.txt` file in this directory:

```
$ cat /tmp/brooklyn-martin/apps/jIzIHxTP/entities/TomcatServer_BUHgQe0s/pid.txt
73714
```

In this case, the PID in the file is 73714. You can then check if the process is running using `ps`. You can also pipe the output to `fold` so the full launch command is visible:

```
$ ps -p 73714 | fold -w 120
PID TTY          TIME CMD
73714 ??          0:08.03 /Library/Java/JavaVirtualMachines/jdk1.8.0_51.jdk/Contents/Home/bin/java -Dnop -Djava.
util.logg
ing.manager=org.apache.juli.ClassLoaderLogManager -javaagent:/tmp/brooklyn-martin/apps/jIzIHxTP/entities/Tomcat
Server_BU
HgQe0s/brooklyn-jmxmp-agent-shaded-0.8.0-SNAPSHOT.jar -Xms200m -Xmx800m -XX:MaxPermSize=400m -Dcom.sun.managemen
t.jmxrem
ote -Dbrooklyn.jmxmp.rmi-port=1099 -Dbrooklyn.jmxmp.port=31001 -Dcom.sun.management.jmxremote.ssl=false -Dcom.s
un.manage
ment.jmxremote.authenticate=false -Djava.endorsed.dirs=/tmp/brooklyn-martin/installs/TomcatServer_7.0.56/apache
-tomcat-7
.0.56/endorsed -classpath /tmp/brooklyn-martin/installs/TomcatServer_7.0.56/apache-tomcat-7.0.56/bin/bootstrap.
jar:/tmp/
brooklyn-martin/installs/TomcatServer_7.0.56/apache-tomcat-7.0.56/bin/tomcat-juli.jar -Dcatalina.base=/tmp/broo
klyn-mart
in/apps/jIzIHxTP/entities/TomcatServer_BUHgQe0s -Dcatalina.home=/tmp/brooklyn-martin/installs/TomcatServer_7.0.
56/apache
-tomcat-7.0.56 -Djava.io.tmpdir=/tmp/brooklyn-martin/apps/jIzIHxTP/entities/TomcatServer_BUHgQe0s/temp org.apac
he.catali
na.startup.Bootstrap start
```

This confirms that the process is running. The next thing we can look at is the `service.notup.indicators` `sensor`. This reads as follows:

```
{"service.process.isRunning":"The software process for this entity does not appear to be running"}
```

This confirms that the problem is indeed due to the `service.process.isRunning` `sensor`. We assumed earlier that this was set by the `isRunning()` method in `TomcatSshDriver.java`, but this isn't always the case. The

`service.process.isRunning` `sensor` is wired up by the `connectSensors()` method in the node's implementation class, in this case `TomcatServerImpl.java`. Tomcat's implementation of `connectSensors()` is as follows:

```
@Override
public void connectSensors() {
    super.connectSensors();

    if (getDriver().isJmxEnabled()) {
        String requestProcessorMbeanName = "Catalina:type=GlobalRequestProcessor,name=\"http-*\"";

        Integer port = isHttpsEnabled() ? getAttribute(HTTPS_PORT) : getAttribute(HTTP_PORT);
        String connectorMbeanName = format("Catalina:type=Connector,port=%s", port);

        JmxWebFeed = JmxFeed.builder()
            .entity(this)
            .period(3000, TimeUnit.MILLISECONDS)
            .pollAttribute(new JmxAttributePollConfig<Integer>(ERROR_COUNT)
                .objectName(requestProcessorMbeanName)
```

```

        .attributeName("errorCount"))
    .pollAttribute(new JmxAttributePollConfig<Integer>(REQUEST_COUNT)
        .objectName(requestProcessorMbeanName)
        .attributeName("requestCount"))
    .pollAttribute(new JmxAttributePollConfig<Integer>(TOTAL_PROCESSING_TIME)
        .objectName(requestProcessorMbeanName)
        .attributeName("processingTime"))
    .pollAttribute(new JmxAttributePollConfig<String>(CONNECTOR_STATUS)
        .objectName(connectorMbeanName)
        .attributeName("stateName"))
    .pollAttribute(new JmxAttributePollConfig<Boolean>(SERVICE_PROCESS_IS_RUNNING)
        .objectName(connectorMbeanName)
        .attributeName("stateName")
        .onSuccess(Functions.forPredicate(Predicates.<Object>equalTo("STARTED"))))
        .setOnFailureOrException(false))
    .build();

    jmxAppFeed = JavaAppUtils.connectMXBeanSensors(this);
} else {
    // if not using JMX
    LOG.warn("Tomcat running without JMX monitoring; limited visibility of service available");
    connectServiceUpIsRunning();
}
}

```

We can see here that if jmx is not enabled, the method will call `connectServiceUpIsRunning()` which will use the default PID-based method of determining if a process is running. However, as JMX is running, the `service.process.isRunning` sensor (denoted here by the `SERVICE_PROCESS_IS_RUNNING` variable) is set to true if and only if the `stateName` JMX attribute equals `STARTED`. We can see from the previous call to `.pollAttribute` that this attribute is also published to the `CONNECTOR_STATUS` sensor. The `CONNECTOR_STATUS` sensor is defined as follows:

```

AttributeSensor<String> CONNECTOR_STATUS =
    new BasicAttributeSensor<String>(String.class, "webapp.tomcat.connectorStatus", "Catalina connector state name");

```

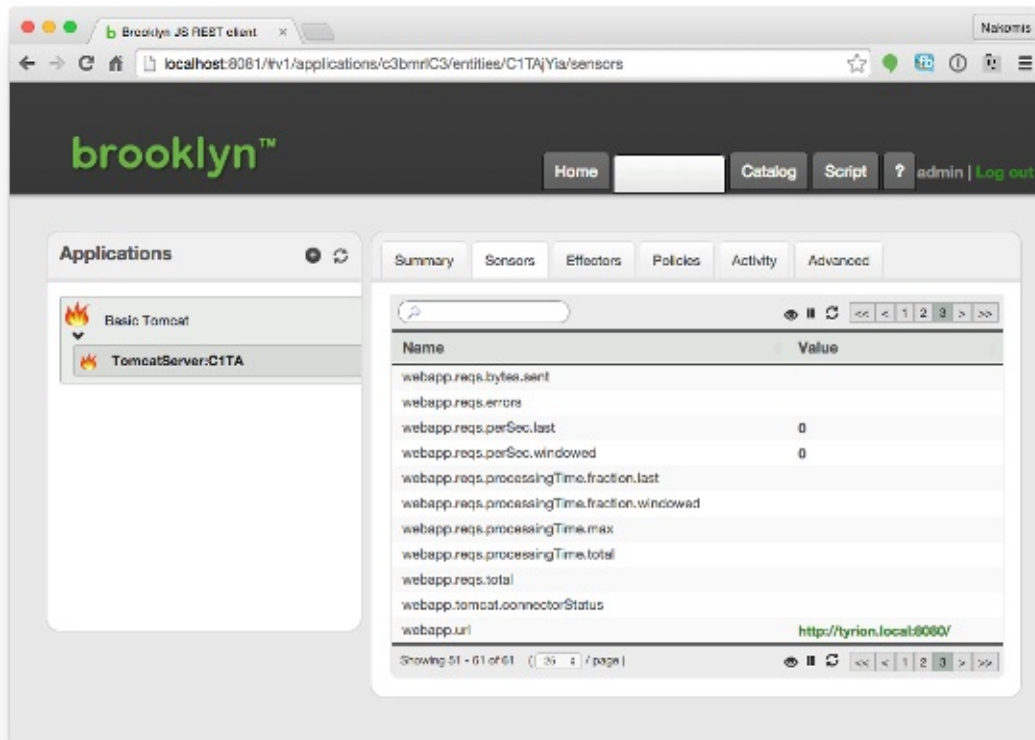
Let's go back to the Brooklyn debug console and look for the `webapp.tomcat.connectorStatus` :

The screenshot shows the Brooklyn JS REST client interface. The browser address bar indicates the URL: `localhost:8081/#v1/applications/c3bmriC3/entities/C1TA/Yia/sensors`. The interface has a dark header with the Brooklyn logo and navigation links: Home, Catalog, Script, and admin | Log out. On the left, the 'Applications' sidebar shows a tree structure with 'Basic Tomcat' and 'TomcatServer:C1TA'. The main panel has tabs for 'Summary', 'Sensors', 'Effectors', 'Policies', 'Activity', and 'Advanced'. The 'Sensors' tab is selected, displaying a table of sensor data. The table has columns 'Name' and 'Value'. A yellow highlight is on the 'Show/hide empty records' icon (a magnifying glass with a slash) in the top right of the table. The table contains the following data:

Name	Value
java.metrics.uptime	22m 39s
jmx.agent.local.path	/tmp/brooklyn-martin/apps/c3bmriC3/entities/TomcatSe
jmx.context	jmxmri
jmx.direct.port	31001
jmx.service.url	servicejmx:jmxmp://tyrion.local:31001
log.location	/tmp/brooklyn-martin/apps/c3bmriC3/entities/TomcatSe
main.url	http://tyrion.local:8080/
rmi.registry.port	1099
run.dir	/tmp/brooklyn-martin/apps/c3bmriC3/entities/TomcatSe
service.isUp	false
service.notUp.indicators	["service.process.isRunning": "The software process for
service.process.isRunning	false
service.state	ON_FIRE
service.state.expected	on-fire @ 1437491870786 / Tue Jul 21 16:17:50 BST 2015
software.service.provisioning.location	["type": "brooklyn.location.Location", "id": "AnwwYh5o"]
tomcat.shutdownport	31880
webapp.deployedWars	[]
webapp.enabledProtocols	["http"]
webapp.reqs.perSec.last	0
webapp.reqs.perSec.windowed	0
webapp.url	http://tyrion.local:8080/

At the bottom of the table, it says 'Showing 26 - 46 of 46 (of 61)' and '1 / page'.

As the **sensor** is not shown, it's likely that it's simply null or not set. We can check this by clicking the "Show/hide empty records" icon (highlighted in yellow above):



We know from previous steps that the installation and launch scripts completed, and we know the process is running, but we can see here that the server is not responding to JMX requests. A good thing to check here would be that the JMX port is not being blocked by iptables, firewalls or security groups (see the [troubleshooting connectivity guide](#)). Let's assume that we've checked that and they're all open. There is still one more thing that Brooklyn can tell us.

Still on the `sensors` tab, let's take a look at the `log.location` sensor:

```
/tmp/brooklyn-martin/apps/c3bmr1C3/entities/TomcatServer_C1TAjYia/logs/catalina.out
```

This is the `location` of Tomcat's own log file. The `location` of the log file will differ from process to process and when writing a custom `entity` you will need to check the software's own documentation. If your `blueprint`'s ssh driver extends `JavaSoftwareProcessSshDriver`, the value returned by the `getLogFileLocation()` method will automatically be published to the `log.location` sensor. Otherwise, you can publish the value yourself by calling `entity.setAttribute(Attributes.LOG_FILE_LOCATION, getLogFileLocation());` in your ssh driver

Note: The log file will be on the server to which you have deployed Tomcat, and not on the Brooklyn server. Let's take a look in the log file:

```
$ less /tmp/brooklyn-martin/apps/c3bmr1C3/entities/TomcatServer_C1TAjYia/logs/catalina.out

Jul 21, 2015 4:12:12 PM org.apache.tomcat.util.digester.Digester fatalError
SEVERE: Parse Fatal Error at line 143 column 3: The element type "unmatched-element" must be terminated by the
matching end-tag "</unmatched-element>".
    org.xml.sax.SAXParseException; systemId: file:/tmp/brooklyn-martin/apps/c3bmr1C3/entities/TomcatServer_C1TA
jYia/conf/server.xml; lineNumber: 143; columnNumber: 3; The element type "unmatched-element" must be terminated
by the matching end-tag "</unmatched-element>".
    at com.sun.org.apache.xerces.internal.util.ErrorHandlerWrapper.createSAXParseException(ErrorHandlerWrapper.
java:203)
    at com.sun.org.apache.xerces.internal.util.ErrorHandlerWrapper.fatalError(ErrorHandlerWrapper.java:177)
```

```
at com.sun.org.apache.xerces.internal.impl.XMLErrorReporter.reportError(XMLErrorReporter.java:441)
at com.sun.org.apache.xerces.internal.impl.XMLErrorReporter.reportError(XMLErrorReporter.java:368)
at com.sun.org.apache.xerces.internal.impl.XMLScanner.reportFatalError(XMLScanner.java:1437)
at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl.scanEndElement(XMLDocumentFragmen
tScannerImpl.java:1749)
at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl$FragmentContentDriver.next(XMLDoc
umentFragmentScannerImpl.java:2973)
at com.sun.org.apache.xerces.internal.impl.XMLDocumentScannerImpl.next(XMLDocumentScannerImpl.java:606)
at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl.scanDocument(XMLDocumentFragments
cannerImpl.java:510)
at com.sun.org.apache.xerces.internal.parsers.XML11Configuration.parse(XML11Configuration.java:848)
at com.sun.org.apache.xerces.internal.parsers.XML11Configuration.parse(XML11Configuration.java:777)
at com.sun.org.apache.xerces.internal.parsers.XMLParser.parse(XMLParser.java:141)
at com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser.parse(AbstractSAXParser.java:1213)
at com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl$JAXPSAXParser.parse(SAXParserImpl.java:649)
at org.apache.tomcat.util.digester.Digester.parse(Digester.java:1561)
at org.apache.catalina.startup.Catalina.load(Catalina.java:615)
at org.apache.catalina.startup.Catalina.start(Catalina.java:677)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:497)
at org.apache.catalina.startup.Bootstrap.start(Bootstrap.java:321)
at org.apache.catalina.startup.Bootstrap.main(Bootstrap.java:455)
Jul 21, 2015 4:12:12 PM org.apache.catalina.startup.Catalina load
WARNING: Catalina.start using conf/server.xml: The element type "unmatched-element" must be terminated by the m
atching end-tag "</unmatched-element>".
Jul 21, 2015 4:12:12 PM org.apache.catalina.startup.Catalina start
SEVERE: Cannot start server. Server instance is not configured.
```

As expected, we can see here that the `unmatched-element` element has not been terminated in the `server.xml` file

Monitoring Memory Usage

Memory Usage

Brooklyn tries to keep in memory as much history of its activity as possible, for displaying through the UI, so it is normal for it to consume as much memory as it can. It uses "soft references" so these objects will be cleared if needed, but **it is not a sign of anything unusual if Brooklyn is using all its available memory**.

The number of active tasks, CPU usage, thread counts, and retention of soft reference objects are a much better indication of load. This information can be found by looking in the log for lines containing `brooklyn gc`, such as:

```
2016-09-16 16:19:43,337 DEBUG o.a.b.c.m.i.BrooklynGarbageCollector [brooklyn-gc]: brooklyn gc (before) - using
910 MB / 3.76 GB memory; 98% soft-reference maybe retention (of 362); 35 threads; tasks: 0 active, 2 unfinished
; 31 remembered, 1013 total submitted)
```

The soft-reference figure is indicative, but the lower this is, the more the JVM has decided to get rid of items that were desired to be kept but optional. It only tracks some soft-references (those wrapped in `Maybe`), and of course if there are many many such items the JVM will have to get rid of some, so a lower figure does not necessarily mean a problem. Typically however if there's no `OutOfMemoryError` (OOM) reported, there's no problem.

Problem Indicators and Resolutions

Two things that *do* normally indicate a problem with memory are:

- `OutOfMemoryError` exceptions being thrown
- Memory usage high *and* CPU high, where the CPU is spent doing full garbage collection

One possible cause is the JVM doing a poorly-selected GC strategy, as described in [Oracle Java bug 6912889](#). This can be confirmed by running the "analyzing soft reference usage" technique below; memory should shrink dramatically then increase until the problem recurs. This can be fixed by passing `-XX:SoftRefLRUPolicyMSPerMB=1` to the JVM, as described in [Brooklyn issue 375](#).

Other common JVM options include `-Xms256m -Xmx1g` (depending on JVM provider and version) to set the right balance of memory allocation. In some cases a larger `-Xmx` value may simply be the fix (but this should not be the case unless many or large blueprints are being used).

If the problem is not with soft references but with real memory usage, the culprit is likely a memory leak, typically in [blueprint](#) design. An early warning of this situation is the "soft-reference maybe retention" level decreasing. In these situations, follow the steps as described below for "Investigating Leaks".

Analyzing Soft Reference Usage

If you are concerned about memory usage, or doing evaluation on test environments, the following method (in the Groovy console) can be invoked to force the system to reclaim as much memory as possible, including *all* soft references:

```
org.apache.brooklyn.util.javalang.MemoryUsageTracker.forceClearSoftReferences()
```

In good situations, memory usage should return to a small level.
This call can be disruptive to the system however so use with care.

The above method can also be configured to run automatically when memory usage is detected to hit a certain level. That can be useful if external policies are being used to warn on high memory usage, and you want to keep some headroom. Many JVM authorities discourage interfering with its garbage collector, however, so use with care and study the particular JVM you are using. See the class `BrooklynGarbageCollector` for more information.

Investigating Leaks

If a memory leak is found, the first place to look should be the WARN/ERROR logs. Many common causes of leaks, including as runaway tasks and cyclic dependent configuration, will show their own log errors prior to the memory error.

You should also note the task counts in the `brooklyn gc` messages described above, and if there are an exceptional number of tasks or tasks are not clearing, other log messages will describe what is happening, and the in-product task view can indicate issues.

Sometimes slow leaks can occur if blueprints do not clean up entities or locations. These can be diagnosed by noting the number of files written to the persistence `location`, if persistence is being used. Deploying then destroying a `blueprint` should not leave anything behind in the persistence directory.

Where problems have been encountered in the past, we have resolved them and/or worked to improve logging and early identification. Please report any issues so that we can improve this further. In many cases we can also give advice on what other log `grep` patterns can be useful.

Standard Java Techniques

Useful standard Java techniques for tracking memory leaks include:

- `jstack <pid>` to see what tasks are running
- `jmap -histo:live <pid>` to see what objects are using memory (see below)
- Memory profilers such as VisualVM or Eclipse MAT, either connected to a running system or against a heap dump generated on an OOME

More information is available on [the Oracle Java web site](#).

Note that some of the above techniques will often include soft and weak references that are irrelevant to the problem (and will be cleared on an OOME). Objects that may be cached in that way include:

- `BasicConfigKey` (used for the web server and many blueprints)
- `DslComponent` and `*Task` (used for Brooklyn activities and dependent configuration)
- `jcclouds` items including `ImageImpl` (to cache data on cloud service providers)

On the other hand any of the above may also indicate a leak. Taking snapshots after a `forceClearSoftReferences()` (above) invocation and comparing those is one technique to filter out noise. Another is to wait until there is an OOME and look just after, because that will clear all non-essential data from memory. (The `forceClearSoftReferences()` actually works by triggering an OOME, in as safe a way as possible.)

If leaked items are found, a profiler will normally let you see their content and walk backwards along their references to find out why they are being retained.

Summary of Techniques

The following sequence of techniques is a common approach to investigating and fixing memory issues:

- Note the log lines about `brooklyn gc` , including memory and tasks
- Do not assume high memory usage alone is an error, as soft reference caches are deliberate; use `forceClearSoftReferences()` to clear these
- Note any WARN/ERROR messages in the log
- Tune JVM memory allocation and GC
- Look for leaking locations or references by creating then destroying a `blueprint`
- Use standard JVM profilers
- Inform the Apache Brooklyn community

Based on the installation method you choose, the paths to the installed components of Apache Brooklyn will be different. The following table will help you to easily locate these:

Installation method	Brooklyn Home	Brooklyn Logs	Brooklyn Configuration
RPM Package	/opt/brooklyn (symlink to /opt/brooklyn-<version>)	/var/log/brooklyn (symlink to /opt/brooklyn-<version>/data/log)	/etc/brooklyn
DEB Package	/opt/brooklyn (symlink to /opt/brooklyn-<version>)	/var/log/brooklyn (symlink to /opt/brooklyn-<version>/data/log)	/etc/brooklyn
Vagrant	/opt/brooklyn (symlink to /opt/brooklyn-<version>)	/var/log/brooklyn (symlink to /opt/brooklyn-<version>/data/log)	/etc/brooklyn
Tarball Zip Package	/path/of/untar/archive	/path/of/untar/archive/data/log	/path/of/untar/archive/etc

Other 0.12.0 Resources

- [Glossary](#)
- [Versions](#)
- [Release Notes](#)

Apache jclouds

An open source Java library that provides a consistent interface to many clouds. Apache Brooklyn uses [Apache jclouds](#) as its core cloud abstraction.

Autonomic

Refers to the self-managing characteristics of distributed computing resources, adapting to unpredictable changes while hiding intrinsic complexity to operators and users.

Blueprint

A description of an application or system, which can be used for its automated deployment and runtime management. The [blueprint](#) describes a model of the application (i.e. its components, their configuration, and their relationships), along with policies for runtime management. The [blueprint](#) can be described in [YAML](#) or Java.

Effector

Effectors are tools Apache Brooklyn provides, that allow you to manipulate the live entities within an application. They are operations applied on entities.

Enricher

Generates new events or [sensor](#) values (metrics) for an [entity](#), usually by aggregating or modifying data from one or more other sensors.

Entity

A component of an application or system. This could be a physical component, a service, a grouping of components, or a logical construct describing part of an application/system. It is a "managed element" in [autonomic](#) computing parlance.

Location

A server or resource to which Apache Brooklyn can deploy applications

Policy

Part of an [autonomic](#) management system, performing runtime management. A [policy](#) is associated with an [entity](#); it normally manages the health of that [entity](#) or an associated group of entities (e.g. HA policies or auto-scaling policies). A [policy](#) performs actions on entities, based on their [sensor](#) values and [policy](#) configuration.

Sensor

A [sensor](#) is a property, or attribute of an Apache Brooklyn [entity](#), updated in real-time.

YAML

A human-readable data format. See the [Wikipedia article](#) for more information.

CAMP and TOSCA

OASIS Cloud Application Management for Platforms (CAMP) and OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) are specifications that aim to standardise the portability and management of cloud applications.

Release Notes

Version 1.0.0-SNAPSHOT

You are viewing a SNAPSHOT release (master branch), so this list is in progress!

Thanks go to our community for their improvements, feedback and guidance, and to Brooklyn's commercial users for funding much of this development.

New Features

Backwards Compatibility

Changes since 0.12.0:

For changes in prior versions, please refer to the release notes for those versions by selecting the one you are interested in on the left menu.

Maven Build

The Basics

The full build requires the following software to be installed:

- Maven (v3.1+)
- Java (v1.7+, 1.8 recommended)
- Go (v1.6+) [if building the CLI client]
- rpm tools [if building the dist packages for those platforms]

With these in place, you should be able to build everything with a:

```
mvn clean install
```

Alternatively you can build most things with just Java and Maven installed using:

```
mvn clean install -Dno-go-client -Dno-rpm
```

Other tips:

- Add `-DskipTests` to skip tests (builds much faster, but not as safe)
- You may need more JVM memory, e.g. at the command-line (or in `.profile`):

```
export MAVEN_OPTS="-Xmx1024m -Xms512m"
```
- Run `-PIntegration` to run integration tests, or `-PLive` to run live tests ([tests described here](#))
- You may need to install `rpm` package to build RPM packages: `brew install rpm` for Mac OS, `apt-get install rpm` for Ubuntu, `yum install rpm` for Centos/RHEL. On Mac OS you may also need to set `%_tmppath /tmp` in `~/.rpmmacros`.
- If you're looking at the maven internals, note that many of the settings are inherited from parent projects (see for instance `brooklyn-server/parent/pom.xml`)
- For tips on building within various IDEs, look [here](#).

When the RAT Bites

We use RAT to ensure that all files are compliant to Apache standards. Most of the time you shouldn't see it or need to know about it, but if it detects a violation, you'll get a message such as:

```
[ERROR] Failed to execute goal org.apache.rat:apache-rat-plugin:0.10:check (default) on project brooklyn-parent : Too many files with unapproved license: 1 See RAT report in: /Users/alex/Data/cloudsoft/dev/gits/brooklyn/target/rat.txt -> [Help 1]
```

If there's a problem, see the file `rat.txt` in the `target` directory of the failed project. (Maven will show you this link in its output.)

Often the problem is one of the following:

- You've added a file which requires the license header but doesn't have it

Resolution: Simply copy the header from another file

- You've got some temporary files which RAT things should have headers

Resolution: Move the files away, add headers, or turn off RAT (see below)

- The project structure has changed and you have stale files (e.g. in a `target` directory)

Resolution: Remove the stale files, e.g. with `git clean -df` (and if needed a `find . -name target -prune -exec rm -rf {} \;` to delete folders named `target`)

To disable RAT checking on a build, set `rat.ignoreErrors` , e.g. `mvn -Drat.ignoreErrors=true clean install` . (But note you will need RAT to pass in order for a PR to be accepted!)

If there is a good reason that a file, pattern, or directory should be permanently ignored, that is easy to add inside the root `pom.xml` .

Other Handy Hints

- On some **Ubuntu** (e.g. 10.4 LTS) maven v3 is not currently available from the repositories. Some instructions for installing at are [at superuser.com](http://superuser.com).
- The **mvnf** script ([get the gist here](#)) simplifies building selected projects, so if you just change something in `software-webapp` and then want to re-run the examples you can do:

```
examples/simple-web-cluster% mvnf ../../{software/webapp,usage/all}
```

Appendix: Sample Output

A healthy build will look something like the following, including a few warnings (which we have looked into and understand to be benign and hard to get rid of them, although we'd love to if anyone can help!):

```
% mvn clean install
[INFO] Scanning for projects...
[INFO] -----
[INFO] Reactor Build Order:
[INFO]
[INFO] Brooklyn REST JavaScript Web GUI
[INFO] Brooklyn Server Root
[INFO] Brooklyn Parent Project
[INFO] Brooklyn Test Support Utilities
[INFO] Brooklyn Logback Includable Configuration
[INFO] Brooklyn Common Utilities
...
[WARNING] Ignoring project type war - supportedProjectTypes = [jar]
...
[WARNING] We have a duplicate org/xmlpull/v1/XmlPullParser.class in ~/.m2/repository/xpp3/xpp3_min/1.1.4c/xpp3_min-1.1.4c.jar
...
[INFO] - maven-assembly-plugin:2.3:single (build-distribution-dir) @ brooklyn-dist -
[INFO] Reading assembly descriptor: src/main/config/build-distribution-dir.xml
{% comment %}BROOKLYN_VERSION{% endcomment %}[WARNING] Cannot include project artifact: io.brooklyn:brooklyn-dist:jar:1.0.0-SNAPSHOT; it doesn't have an associated file or directory.
[INFO] Copying files to ~/repos/apache/brooklyn/usage/dist/target/brooklyn-dist
```



```

[WARNING] Assembly file: ~/repos/apache/brooklyn/usage/dist/target/brooklyn-dist is not a regular file (it may
be a directory). It cannot be attached to the project build for installation or deployment.

...

[INFO] - maven-assembly-plugin:2.3:single (build-distribution-archive) @ brooklyn-dist -
[INFO] Reading assembly descriptor: src/main/config/build-distribution-archive.xml
{% comment %}BROOKLYN_VERSION{% endcomment %}[WARNING] Cannot include project artifact: io.brooklyn:brooklyn-di
st:jar:1.0.0-SNAPSHOT; it doesn't have an associated file or directory.
{% comment %}BROOKLYN_VERSION{% endcomment %}[INFO] Building tar: /Users/aled/repos/apache/brooklyn/usage/dist/
target/brooklyn-1.0.0-SNAPSHOT-dist.tar.gz
{% comment %}BROOKLYN_VERSION{% endcomment %}[WARNING] Cannot include project artifact: io.brooklyn:brooklyn-di
st:jar:1.0.0-SNAPSHOT; it doesn't have an associated file or directory.

...

[WARNING] Don't override file /Users/aled/repos/apache/brooklyn/usage/archetypes/quickstart/target/test-classes
/projects/integration-test-1/project/brooklyn-sample/src/main/resources/sample-icon.png

...

[INFO] Reactor Summary:
[INFO]
[INFO] Brooklyn Parent Project ..... SUCCESS [3.072s]
[INFO] Brooklyn Utilities to Support Testing (listeners etc) SUCCESS [3.114s]
[INFO] Brooklyn Logback Includable Configuration ..... SUCCESS [0.680s]
[INFO] Brooklyn Common Utilities ..... SUCCESS [7.263s]
[INFO] Brooklyn Groovy Utilities ..... SUCCESS [5.193s]
[INFO] Brooklyn API ..... SUCCESS [2.146s]
[INFO] Brooklyn Test Support ..... SUCCESS [2.517s]
[INFO] CAMP Server Parent Project ..... SUCCESS [0.075s]
[INFO] CAMP Base ..... SUCCESS [4.079s]
[INFO] Brooklyn REST Swagger Apidoc Utilities ..... SUCCESS [1.983s]
[INFO] Brooklyn Logback Configuration ..... SUCCESS [0.625s]
[INFO] CAMP Server ..... SUCCESS [5.446s]
[INFO] Brooklyn Core ..... SUCCESS [1:24.122s]
[INFO] Brooklyn Policies ..... SUCCESS [44.425s]
[INFO] Brooklyn Hazelcast Storage ..... SUCCESS [7.143s]
[INFO] Brooklyn Jclouds Location Targets ..... SUCCESS [16.488s]
[INFO] Brooklyn Secure JMXMP Agent ..... SUCCESS [8.634s]
[INFO] Brooklyn JMX RMI Agent ..... SUCCESS [2.315s]
[INFO] Brooklyn Software Base ..... SUCCESS [28.538s]
[INFO] Brooklyn Network Software Entities ..... SUCCESS [3.896s]
[INFO] Brooklyn OSGi Software Entities ..... SUCCESS [4.589s]
[INFO] Brooklyn Web App Software Entities ..... SUCCESS [17.484s]
[INFO] Brooklyn Messaging Software Entities ..... SUCCESS [7.106s]
[INFO] Brooklyn Database Software Entities ..... SUCCESS [5.229s]
[INFO] Brooklyn NoSQL Data Store Software Entities ..... SUCCESS [11.901s]
[INFO] Brooklyn Monitoring Software Entities ..... SUCCESS [4.027s]
[INFO] Brooklyn CAMP REST API ..... SUCCESS [15.285s]
[INFO] Brooklyn REST API ..... SUCCESS [4.489s]
[INFO] Brooklyn REST Server ..... SUCCESS [30.270s]
[INFO] Brooklyn REST Client ..... SUCCESS [7.007s]
[INFO] Brooklyn REST JavaScript Web GUI ..... SUCCESS [24.397s]
[INFO] Brooklyn Launcher ..... SUCCESS [15.923s]
[INFO] Brooklyn Command Line Interface ..... SUCCESS [9.279s]
[INFO] Brooklyn All Things ..... SUCCESS [13.875s]
[INFO] Brooklyn Distribution ..... SUCCESS [49.370s]
[INFO] Brooklyn Quick-Start Project Archetype ..... SUCCESS [12.053s]
[INFO] Brooklyn Examples Aggregator Project ..... SUCCESS [0.085s]
[INFO] Brooklyn Examples Support Aggregator Project - Webapps SUCCESS [0.053s]
[INFO] hello-world-webapp Maven Webapp ..... SUCCESS [0.751s]
[INFO] hello-world-sql-webapp Maven Webapp ..... SUCCESS [0.623s]
[INFO] Brooklyn Simple Web Cluster Example ..... SUCCESS [5.398s]
[INFO] Brooklyn Global Web Fabric Example ..... SUCCESS [3.176s]
[INFO] Brooklyn Simple Messaging Publish-Subscribe Example SUCCESS [3.217s]
[INFO] Brooklyn NoSQL Cluster Examples ..... SUCCESS [6.790s]
[INFO] Brooklyn QA ..... SUCCESS [7.117s]
[INFO] -----

```

```
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 8:33.983s
[INFO] Finished at: Mon Jul 21 14:56:46 BST 2014
[INFO] Final Memory: 66M/554M
[INFO] -----
```

IDE Setup

Gone are the days when IDE integration always just works... Maven and Eclipse fight, neither quite gets along perfectly with Groovy, git branch switches (sooo nice) can be slow, etc etc.

But with a bit of a dance the IDE can still be your friend, making it much easier to run tests and debug.

As a general tip, don't always trust the IDE to build correctly; if you hit a snag, do a command-line `mvn clean install` (optionally with `-DskipTests` and/or `-Dno-go-client`) then refresh the project.

See instructions below for specific IDEs.

Eclipse

The default Eclipse downloads already include all of the plugins needed for working with the Brooklyn project. Optionally you can install the Groovy and TestNG plugins, but they are not required for building the project. You can install these using Help -> Install New Software, or from the Eclipse Marketplace:

- Groovy Plugin: GRECLIPSE from dist.springsource.org/snapshot/GRECLIPSE/e4.5/; Be sure to include Groovy 2.3 compiler support and Maven-Eclipse (m2e) support. More details including download sites for other versions can be found at the [Groovy Eclipse Plugin site](#).
- TestNG Plugin: beust TestNG from beust.com/eclipse

As of this writing, Eclipse 4.5 and Eclipse 4.4 are commonly used, and the codebase can be imported (Import -> Existing Maven Projects) and successfully built and run inside an IDE. However there are quirks, and mileage may vary. Disable `Build Automatically` from the `Project` menu if the IDE is slow to respond.

If you encounter issues, the following hints may be helpful:

- If m2e reports import problems, it is usually okay (even good) to mark all to "Resolve All Later". The build-helper connector is useful if you're prompted for it, but do *not* install the Tycho OSGi configurator (this causes show-stopping IAE's, and we don't need Eclipse to make bundles anyway). You can manually mark as permanently ignored certain errors; this updates the pom.xml (and should be current).
- A quick command-line build (`mvn clean install -DskipTests -Dno-go-client`) followed by a workspace refresh can be useful to re-populate files which need to be copied to `target/`
- m2e likes to put `excluding="**"` on `resources` directories; if you're seeing funny missing files (things like not resolving jclouds/aws-ec2 locations or missing WARs), try building clean install from the command-line then doing Maven -> Update Project (clean uses a maven-replacer-plugin to fix `.classpath` s). Alternatively you can go through and remove these manually in Eclipse (Build Path -> Configure) or the filesystem, or use the following command to remove these rogue blocks in the generated `.classpath` files:

```
% find . -name .classpath -exec sed -i.bak 's/[ ]*..cluding="[\*\~]*\(\.java\)*/g' {} \;
```

- You may need to ensure `src/main/{java,resources}` is created in each project dir, if (older versions) complain about missing directories, and the same for `src/test/{java,resources}` if there are tests (`src/test` exists):

```
find . \( -path "**/src/main" -or -path "**/src/test" \) -exec echo {} \; -exec mkdir -p {}/{java,resources} \;
```

If the pain starts to be too much, come find us on IRC #brooklyncentral or [elsewhere](#) and we can hopefully share our pearls. (And if you have a tip we haven't mentioned please let us know that too!)

IntelliJ IDEA

To develop or debug Brooklyn in IntelliJ, you will need to ensure that the Groovy and TestNG plugins are installed via the IntelliJ IDEA | Preferences | Plugins menu. Once installed, you can open Brooklyn from the root folder, (e.g.

```
~/myfiles/brooklyn )
```

 which will automatically open the subprojects.

Brooklyn has informally standardized on arranging `import` statements as per Eclipse's default configuration. IntelliJ's default configuration is different, which can result in unwanted "noise" in commits where imports are shuffled backward and forward between the two types - PRs which do this will likely fail the review. To avoid this, reconfigure IntelliJ to organize imports similar to Eclipse. See [this StackOverflow answer](#) for a suitable configuration.

Netbeans

Tips from Netbeans users wanted!

Debugging Tips

To debug Brooklyn, create a launch configuration which launches the `BrooklynJavascriptGuiLauncher` class. NOTE: You may need to add additional projects or folders to the classpath of the run configuration (e.g. add the brooklyn-software-nosql project if you wish to deploy a MongoDBServer). You will also need to ensure that the working directory is set to the jsgui folder. For IntelliJ, you can set the 'Working directory' of the Run/Debug Configuration to `$MODULE_DIR$/../jsgui`. For Eclipse, use the default option of `${workspace_loc:brooklyn-jsgui}`.

To debug the jsgui (the Brooklyn web console), you will need to build Brooklyn with `-DskipOptimization` to prevent the build from minifying the javascript. When building via the command line, use the command `mvn clean install -DskipOptimization`, and if you are using IntelliJ IDEA, you can add the option to the Maven Runner by clicking on the Maven Settings icon in the Maven Projects tool window and adding the `skipOptimization` property with no value.

When running at the command line you can enable remote connections so that one can attach a debugger to the Java process: Run Java with the following on the command line or in `JAVA_OPTS`: `-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=5005`

To debug a brooklyn instance that has been run with the above `JAVA_OPTS`, create a remote build configuration (IntelliJ - Run | Edit Configurations | + | Remote) with the default options, ensuring the port matches the address specified in `JAVA_OPTS`.

Code Structure

Brooklyn is split into the following subprojects:

- **brooklyn-server:**
 - **api:** the pure-Java interfaces for interacting with the system
 - **camp:** the components for a server which speaks with the CAMP REST API and understands the CAMP **YAML** plan language
 - **core:** the base class implementations for entities and applications, **entity** traits, locations, policies, **sensor** and **effector** support, tasks, and more
 - **karaf:** OSGi support
 - **launcher:** for launching brooklyn, either using a main method or invoked from the cli project
 - **locations:** specific **location** integrations
 - **jclouds:** integration with many cloud APIs and providers via **Apache jclouds**
 - **logging:** how we enable configurable logging
 - **logback-includes:** Various helpful logback XML files that can be included; does not contain logback.xml
 - **logback-xml:** Contains a logback.xml that references the include files in brooklyn-logback-includes
 - **parent:** a meta-project parent to collect dependencies and other maven configuration for re-use
 - **policy:** collection of useful policies for automating **entity** activity
 - **rest:** supporting the REST API
 - **rest-api:** The API classes for the Brooklyn REST api
 - **rest-client:** A client Java implementation for using the Brooklyn REST API
 - **rest-server:** The server-side implementation of the Brooklyn REST API
 - **server-cli:** implementation of the Brooklyn *server* command line interface; not to be confused with the client CLI
 - **software:** support frameworks for creating entities which mainly launch software processes on machines
 - **base:** software process lifecycle abstract classes and drivers (e.g. SSH)
 - **winrm:** support for connecting to Windows machines
 - **test-framework:** provides Brooklyn entities for building **YAML** tests for other entities
 - **test-support:** provides Brooklyn-specific support for Java TestNG tests, used by nearly all projects in scope
`test` , building on `utils/test-support`
 - **utils:** projects with lower level utilities
 - **common:** Utility classes and methods developed for Brooklyn but not dependent on Brooklyn
 - **groovy:** Groovy extensions and utility classes and methods developed for Brooklyn but not dependent on Brooklyn
 - **jmx/jmxmp-ssl-agent:** An agent implementation that can be attached to a Java process, to give expose secure JMXMP
 - **jmx/jmxrmi-agent:** An agent implementation that can be attached to a Java process, to give expose JMX-RMI without requiring all high-number ports to be open
 - **rest-swagger:** Swagger REST API utility classes and methods developed for Brooklyn but not dependent on Brooklyn
 - **test-support:** Test utility classes and methods developed for Brooklyn but not dependent on Brooklyn
- **brooklyn-ui:** Javascript web-app for the brooklyn management web console (builds a WAR)
- **brooklyn-library:** a library of useful blueprints
 - **examples:** some canonical examples
 - **qa:** longevity and stress tests
 - **sandbox:** experimental items
 - **software:** blueprints for software processes
 - **webapp:** web servers (JBoss, Tomcat), load-balancers (Nginx), and DNS (Geoscaling)

- **database**: relational databases (SQL)
- **nosql**: datastores other than RDBMS/SQL (often better in distributed environments)
- **messaging**: messaging systems, including Qpid, Apache MQ, RabbitMQ
- **monitoring**: monitoring tools, including Monit
- **osgi**: OSGi servers
- **brooklyn-docs**: the markdown source code for this documentation
- **brooklyn-dist**: projects for packaging Brooklyn and making it easier to consume

```
* **all**: maven project to supply a shaded JAR (containing all dependencies) for convenience
* **archetypes**: A maven archetype for easily generating the structure of new downstream projects
* **dist**: builds brooklyn as a downloadable .zip and .tar.gz
* **scripts**: various scripts useful for building, updating, etc. (see comments in the scripts)
```

Tests

We have the following tests groups:

- normal (i.e. no group) -- should run quickly, not need internet, and not side effect the machine (apart from a few /tmp files)
- Integration -- deploys locally, may read and write from internet, takes longer.

```
If you change an entity, rerun the relevant integration test to make sure all is well!
```

- Live -- deploys remotely, may provision machines (but should clean up, getting rid of them in a try block)
- Live-sanity -- a sub-set of "Live" that can be run regularly; a trade-off of optimal code coverage for the time/cost of those tests.
- WIP -- short for "work in progress", this will disable the test from being run by the normal brooklyn maven profiles, while leaving the test enabled so that one can work on it in IDEs or run the selected test(s) from the command line.
- Acceptance -- this (currently little-used) group is for very long running tests, such as soak tests

To run these from the command line, use something like the following:

- normal: `mvn clean install`
- integration: `mvn clean verify -PEssentials,Locations,Entities,Integration -Dmaven.test.failure.ignore=true --fail-never`
- Live: `mvn clean verify -PEntities,Locations,Entities,Live -Dmaven.test.failure.ignore=true --fail-never`
- Live-sanity: `mvn clean verify -PEntities,Locations,Entities,Live-sanity -Dmaven.test.failure.ignore=true --fail-never`

To run a single test, use something like the following:

- run a single test class: `mvn -Dtest=org.apache.brooklyn.enricher.stock.EnrichersTest -DfailIfNoTests=false test`
- run a single test method: `mvn -Dtest=org.apache.brooklyn.enricher.stock.EnrichersTest#testAdding -DfailIfNoTests=false test`

License Considerations

The Apache Software Foundation, quite rightly, place a high standard on code provenance and license compliance. The Apache license is flexible and compatible with many other types of license, meaning there is generally little problem with incorporating other open source works into Brooklyn (with GPL being the notable exception). However diligence is required to ensure that the project is legally sound, and third parties are rightfully credited where appropriate.

This page is an interpretation of the [Apache Legal Previously Asked Questions](#) page as it specifically applies to the Brooklyn project, such as how we organise our code and the releases that we make. However this page is not authoritative; if there is any conflict between this page and the Previously Asked Questions or other Apache Legal authority, they will take precedence over this page.

If you have any doubt, please ask on the Brooklyn mailing list, and/or the Apache Legal mailing list.

What code licenses can we bundle?

Apache Legal maintains the "[Category A](#)" list, which is a list of licenses that are compatible with the Apache License; that is, code under these licenses can be imported into Brooklyn without violating Brooklyn's Apache License nor the code's original license (subject to correctly modifying the `LICENSE` and/or `NOTICE` files; see below).

Apache Legal also maintain the "[Category X](#)" list. Code licensed under a Category X license **cannot** be imported into Brooklyn without violating either Brooklyn's Apache license or the code's original license.

There is also a "[Category B](#)" list, which are licenses that are compatible with the Apache license only under certain circumstances. In practice, this means that we can declare a dependency on a library licensed under a Category B license, and bundle the binary build of the library in our binary builds, but we cannot import its source code into the Brooklyn codebase.

If the code you are seeking to import does not appear on any of these lists, check to see if the license content is the same as a known license. For example, many projects actually use a BSD license but do not label it as "The BSD License". If you are still not certain about the license, please ask on the Brooklyn mailing list, and/or the Apache Legal mailing list.

About LICENSE and NOTICE files

Apache Legal requires that *each* artifact that the project releases contains a `LICENSE` and `NOTICE` file that is *accurate for the contents of that artifact*. This means that, potentially, **every artifact that Brooklyn releases may contain a different `LICENSE` and `NOTICE` file**. In practice, it's not usually that complicated and there are only a few variations of these files needed.

Furthermore, *accurate* `LICENSE` and `NOTICE` files means that it correctly attributes the contents of the artifact, and it does not contain anything unnecessary. This provision is what prevents us creating a mega `LICENSE` file and using it in every single artifact we release, because in many cases it will contain information that is not relevant to an artifact.

What is a correct `LICENSE` and `NOTICE` file?

- A correct `LICENSE` file is one that contains the text of the licence of any part of the code. The Apache Software License V2 will naturally be the first part of this file, as it's the license which we use for all the original code in Brooklyn. If some *Category A* licensed third-party code is bundled with this artifact, then the `LICENSE` file should identify what the third-party code is, and include a copy of its license. For example, if jquery is bundled with a web

app, the `LICENSE` file would include a note jquery.js, its copyright and its license (MIT), and include a full copy of the MIT license.

- A correct `NOTICE` file contains notices required by bundled third-party code above and beyond that which we have already noted in `LICENSE`. In practice modifying `NOTICE` is rarely required beyond the initial note about Apache Brooklyn. See [What Are Required Third-party Notices?](#) for more information

Applying LICENSE and NOTICE files to Brooklyn

When the Brooklyn project makes a release, we produce and release the following types of artifacts:

1. One source release artifact
2. One binary release artifact
3. A large number of Maven release artifacts

Therefore, our source release, our binary release, and every one of our Maven release artifacts, must **each** have their own, individually-tailored, `LICENSE` and `NOTICE` files.

To some extent, this is automated, using scripts in `usage/dist/licensing`; but this must be manually run, and wherever source code is included or a project has insufficient information in its POM, you'll need to add project-specific metadata (with a project-specific `source-inclusions.yaml` file and/or in the dist project's `overrides.yaml` file). See the `README.md` in that project's folder for more information.

Maven artifacts

Each Maven module will generally produce a JAR file from code under `src/main`, and a JAR file from code under `src/test`. (There are some exceptions which may produce different artifacts.)

If the contents of the module are purely Apache Brooklyn original code, and the outputs are JAR files, then *no action is required*. The default build process will incorporate a general-purpose `LICENSE` and `NOTICE` file into all built JAR files. `LICENSE` will contain just a copy of the Apache Software License v2, and `NOTICE` will contain just the module's own notice fragment.

However you will need to take action if either of these conditions are true:

- the module produces an artifact that is **not** a JAR file - for example, the `jsgui` project produces a WAR file;
- the module bundles third-party code that requires a change to `LICENSE` and/or `NOTICE`.

In this case you will need to disable the automatic insertion of `LICENSE` and `NOTICE` and insert your own versions instead.

For an example of a JAR file with customized `LICENSE` / `NOTICE` files, refer to the `usage/cli` project. For an example of a WAR file with customized `LICENSE` / `NOTICE` files, refer to the `usage/jsgui` project.

The source release

In practice, the source release contains nothing that isn't in the individual produced Maven artifacts (the obvious difference about it being source instead of binary isn't relevant). Therefore, the source release `LICENSE` and `NOTICE` can be considered to be the union of every Maven artifact's `LICENSE` and `NOTICE`. The amalgamated files are kept in the root of the repository.

The binary release

This is the trickiest one to get right. The binary release includes everything that is in the source and Maven releases, **plus every Java dependency of the project**. This means that the binary release is pulling in many additional items, each of which have their own license, and which will therefore impact on `LICENSE` and `NOTICE` .

Therefore you must inspect every file that is present in the binary distribution, ascertain its license status, and ensure that `LICENSE` and `NOTICE` are correct.

Miscellaneous Tips and Tricks

General Good Ways of Working

- If working on something which could be contributed to Brooklyn, do it in a project under the `sandbox` directory. This means we can accept pulls more easily (as sandbox items aren't built as part of the main build) and speed up collaboration.
- When debugging an `entity`, make sure the `brooklyn.SSH logger` is set to `DEBUG` and accessible.
- Use tests heavily! These are pretty good to run in the IDE (once you've completed `IDE setup`), and far quicker to spot problems than runtime, plus we get early-warning of problems introduced in the future. (In particular, Groovy's laxity with compilation means it is easy to introduce silly errors which good test coverage will find much faster.)
- If you hit inexplicable problems at runtime, try clearing your Maven caches, or the brooklyn-relevant parts, under `~/.m2/repository`. Also note your IDE might be recompiling at the same time as a Maven command-line build, so consider turning off auto-build.
- When a class or method becomes deprecated, always include `@deprecated` in the Javadoc e.g. "`@deprecated since 0.7.0; instead use {@link ...}`".
 - Include when it was deprecated
 - Suggest what to use instead -- e.g. link to alternative method, and/or code snippet, etc.
 - Consider logging a warning message when a deprecated method or config option is used, saying who is using it (e.g. useful if deprecated config keys are used in `yaml`) -- if it's a method which might be called a lot, some convenience for "warn once per `entity`" would be helpful
 - See the [Java deprecation documentation](#)

Entity Design Tips

- Look at related entities and understand what they've done, in particular which sensors and config keys can be re-used. (Many are inherited from interfaces, where they are declared as constants, e.g. `Attributes` and `usesJmx`.)
- Understand the `location` hierarchy: software process entities typically get an `SshMachineLocation`, and use a `*SshDriver` to do what they need. This will usually have a `MachineProvisioningLocation` parent, e.g. a `JcloudsLocation` (e.g. AWS eu-west-1 with credentials) or possibly a `LocalhostMachineProvisioningLocation`. Clusters will take such a `MachineProvisioningLocation` (or a singleton list); fabircs take a list of locations. Some PaaS systems have their own `location` model, such as `OpenShiftLocation`.
- Finally, don't be shy about `talking with others`, that's far better than spinning your wheels (or worse, having a bad experience), plus it means we can hopefully improve things for other people!

YAML Blueprint Debugging

- Brooklyn will reject any `YAML blueprint` that contains syntax errors and will alert the user of such errors.
- However, it is possible to create a `blueprint` that is syntactically legal but results in runtime problems for Brooklyn (for example, if an `enricher`'s `enricher.producer` value is not immediately resolvable).

- If Brooklyn appears to freeze after deploying a `blueprint`, run the `jstack <brooklyn-pid>` command to view the state of all running threads. By examining this output, it may become obvious which thread(s) are causing the problem, and the details of the stack trace will provide insight into which part of the `blueprint` is incorrectly written.

Project Maintenance

- Adding a new project may need updates to `/pom.xml` `modules` section and `usage/all` dependencies
- Adding a new example project may need updates to `/pom.xml` and `/examples/pom.xml` (and the documentation too!)

Logging

Logging: A Quick Overview

For logging, we use **logback** which implements the slf4j API. This means you can use any slf4j compliant logging framework, with a default configuration which just works out of the box and bindings to the other common libraries (`java.util.logging` , `log4j` , ...) if you prefer one of those.

OSGi based Apache Brooklyn

While developing it may be useful to change logging level of some of the Apache Brooklyn modules. The easiest way to do that is via the karaf console which can be started by `bin/client` . (Details regarding using [Apache Brooklyn Karaf console](#)) For example if you would like to inspect jclouds API calls, enable jclouds.wire logging just enable it from karaf client.

```
log:set DEBUG jclouds.wire
```

To check other log levels.

```
log:list
```

If for some reason log level needs modified before the first start of Karaf then you can modify the config file `etc/org.ops4j.pax.logging.cfg` before hand. For more information check <https://ops4j1.jira.com/wiki/display/paxlogging/Configuration>.

Classic - non-OSGi based Apache Brooklyn

To use:

- **Users:** If using a brooklyn binary installation, simply edit the `logback.xml` or `logback-custom.xml` supplied in the archive, sometimes in a `conf/` directory.
- **Developers:** When setting up a new project, if you want logging it is recommended to include the `brooklyn-logback.xml` project as an *optional* and *provided* maven dependency, and then to put custom logging configuration in either `logback-custom.xml` or `logback-main.xml` , as described below.

Customizing Your Logging

The project `brooklyn-logback.xml` supplies a `logback.xml` configuration, with a mechanism which allows it to be easily customized, consumed, and overridden. You may wish to include this as an *optional* dependency so that it is not forced upon downstream projects. This `logback.xml` file supplied contains just one instruction, to include `logback-main.xml` , and that file in turn includes:

- `logback-custom.xml`
- `brooklyn/logback-appender-file.xml`
- `brooklyn/logback-appender-stdout.xml`
- `brooklyn/logback-logger-excludes.xml`
- `brooklyn/logback-debug.xml`

For the most common customizations, simply create a `logback-custom.xml` on your classpath (ensuring it is loaded *before* brooklyn classes in classpath ordering in the pom) and supply your customizations there:

```
<included>
  <!-- filename to log to -->
  <property name="logging.basename" scope="context" value="acme-app" />

  <!-- additional loggers -->
  <logger name="com.acme.app" level="DEBUG"/>
</included>
```

For other configuration, you can override individual files listed above. For example:

- To remove debug logging, create a trivial `brooklyn/logback-debug.xml`, containing simply `<included/>`.
- To customise stdout logging, perhaps to give it a threshold WARN instead of INFO, create a `brooklyn/logback-appender-stdout.xml` which defines an appender STDOUT.
- To discard all brooklyn's default logging, create a `logback-main.xml` which contains your configuration. This should look like a standard logback configuration file, except it should be wrapped in `<included>` XML tags rather than `<configuration>` XML tags (because it is included from the `logback.xml` which comes with `brooklyn-logback.xml`.)
- To redirect all jclouds logging to a separate file include `brooklyn/logback-logger-debug-jclouds.xml`. This redirects all logging from `org.jclouds` and `jclouds` to one of two files: anything logged from Brooklyn's persistence thread will end up in a `persistence.log`, everything else will end up in `jclouds.log`.

You should **not** supply your own `logback.xml` if you are using `brooklyn-logback.xml`. If you do, logback will detect multiple files with that name and will scream at you. If you wish to supply your own `logback.xml`, do **not** include `brooklyn-logback.xml`. (Alternatively you can include a `logback.groovy` which causes logback to ignore `logback.xml`.)

You can set a specific logback config file to use with:

```
-Dlogback.configurationFile=/path/to/config.xml
```

Assemblies

When building an assembly, it is recommended to create a `conf/logback.xml` which simply includes `logback-main.xml` (which comes from the classpath). Users of the assembly can then edit the `logback.xml` file in the usual way, or they can plug in to the configuration mechanisms described above, by creating files such as `logback-custom.xml` under `conf/`.

Including `brooklyn-logback.xml` as an *optional* and *provided* dependency means everything should work correctly in IDE's but it will not include the extra `logback.xml` file in the assembly. (Alternatively if you include the `conf/` dir in your IDE build, you should exclude this dependency.)

With this mechanism, you can include `logback-custom.xml` and/or other files underneath `src/main/resources/` of a project, as described above (for instance to include custom logging categories and define the log file name) and it should get picked up, both in the IDE and in the assembly.

Tests

Brooklyn projects `test` scope includes the `brooklyn-utils-test-support` project which supplies a `logback-test.xml`. logback uses this file in preference to `logback.xml` when available (ie when running tests). However the `logback-test.xml` Brooklyn uses includes the same `logback-main.xml` call path above, so your configurations should still work.

The only differences of the `logback-test.xml` configuration is that:

- Debug logging is included for all Brooklyn packages

- The log file is called `brooklyn-tests.log`

Caveats

- logback uses SLF4J version 1.6 which is **not compatible** with 1.5.x. If you have dependent projects using 1.5.x (such as older Grails) things may break.
- If you're not getting the logging you expect in the IDE, make sure `src/main/resources` is included in the classpath. (In eclipse, right-click the project, the Build Path -> Configure, then make sure all dirs are included (All) and excluded (None) -- `mvn clean install` should do this for you.)
- You may find that your IDE logs to a file `brooklyn-tests.log` if it doesn't distinguish between test build classpaths and normal classpaths.
- Logging configuration using file overrides such as this is very sensitive to classpath order. To get a separate `brooklyn-tests.log` file during testing, for example, the `brooklyn-test-support` project with scope `test` must be declared as a dependency *before* `brooklyn-logback-includes`, due to the way both files declare `logback-appender-file.xml`.
- Similarly note that the `logback-custom.xml` file is included *after* logging categories and levels are declared, but before appenders are declared, so that logging levels declared in that file dominate, and that properties from that file apply to appenders.
- Finally remember this is open to improvement. It's the best system we've found so far but we welcome advice. In particular if it could be possible to include files from the classpath with wildcards in alphabetical order, we'd be able to remove some of the quirks listed above (though at a cost of some complexity!).

Brooklyn Remote Debugging

Usually during development, you will be running Brooklyn from your IDE (see [IDE Setup](#)), in which case debugging is as simple as setting a breakpoint. There may however be times when you need to debug an existing remote Brooklyn instance (often referred to as Resident Brooklyn, or rBrooklyn) on another machine, usually in the cloud.

Thankfully, the tools are available to do this, and setting it up is quite straightforward. The steps are as follows:

- [Getting the right source code version](#)
- [Starting Brooklyn with a debug listener](#)
- [Creating an SSH tunnel](#)
- [Connecting your IDE](#)

Getting the right source code version

The first step is to ensure that your local copy of the source code is at the version used to build the remote Brooklyn instance. The git commit that was used to build Brooklyn is available via the REST API:

```
http://<remote-address>:<remote-port>/v1/server/version
```

This should return details of the build as a JSON string similar to the following (formatted for clarity):

```
{
  "version": "1.0.0-SNAPSHOT", // BROOKLYN_VERSION
  "buildSha1": "c0fdc15291702281acdebf1b11d431a6385f5224",
  "buildBranch": "UNKNOWN"
}
```

The value that we're interested in is `buildsha1`. This is the git commit that was used to build Brooklyn. We can now checkout and build the Brooklyn code at this commit by running the following in the root of your Brooklyn repo:

```
% git checkout c0fdc15291702281acdebf1b11d431a6385f5224
% mvn clean install -DskipTests
```

Whilst building the code isn't strictly necessary, it can help prevent some IDE issues.

Starting Brooklyn with a debug listener

By default, Brooklyn does not listen for a debugger to be attached, however this behaviour can be set by setting `JAVA_OPTS`, which will require a restart of the Brooklyn node. To do this, SSH to the remote Brooklyn node and run the following in the root of the Brooklyn installation:

```
# NOTE: Running this kill command will lose existing apps and machines if persistence is disabled.
% kill `cat pid_java`
% export JAVA_OPTS="-Xms256m -Xmx1g -agentlib:jdwp=transport=dt_socket,address=127.0.0.1:8888,server=y,suspend=
n"
% bin/brooklyn launch &
```

If `JAVA_OPTS` is not set, Brooklyn will automatically set it to `"-Xms256m -Xmx1g"`, which is why we have prepended the agentlib settings with these values here.

You should see the following in the console output:

```
Listening for transport dt_socket at address: 8888
```

This will indicate the Brooklyn is listening on port 8888 for a debugger to be attached.

Creating an SSH tunnel

If port 8888 is accessible on the remote Brooklyn server, then you can skip this step and simply use the address of the server in place of 127.0.0.1 in the [Connecting your IDE](#) section below. It will normally be possible to make the port accessible by configuring Security Groups, iptables, endpoints etc., but for a quick ad-hoc connection it's usually simpler to create an SSH tunnel. This will create an open SSH connection that will redirect traffic from a port on a local interface via SSH to a port on the remote machine. To create the tunnel, run the following on your local machine:

```
# replace this with the address or IP of the remote Brooklyn node
REMOTE_HOST=<remote-address>
# if you wish to use a different port, this value must match the port specified in the JAVA_OPTS
REMOTE_PORT=8888
# if you wish to use a different local port, this value must match the port specified in the IDE configuration
LOCAL_PORT=8888
# set this to the login user you use to SSH to the remote Brooklyn node
SSH_USER=root
# The private key file used to SSH to the remote node. If you use a password, see the alternative command below
PRIVATE_KEY_FILE=~/.ssh/id_rsa

% ssh -YNf -i $PRIVATE_KEY_FILE -l $SSH_USER -L $LOCAL_PORT:127.0.0.1:$REMOTE_PORT $REMOTE_HOST
```

If you use a password to SSH to the remote Brooklyn node, simply remove the `-i $PRIVATE_KEY_FILE` section like so:

```
ssh -YNf -l $SSH_USER -L $LOCAL_PORT:127.0.0.1:$REMOTE_PORT $REMOTE_HOST
```

If you are using a password to connect, you will be prompted to enter your password to connect to the remote node upon running the SSH command.

The SSH tunnel should now be redirecting traffic from port 8888 on the local 127.0.0.1 network interface via the SSH tunnel to port 8888 on the remote 127.0.0.1 interface. It should now be possible to connect the debugger and start debugging.

Connecting your IDE

Setting up your IDE will differ depending upon which IDE you are using. Instructions are given here for Eclipse and IntelliJ, and have been tested with Eclipse Luna and IntelliJ Ultimate 14.

Eclipse Setup

To debug using Eclipse, first open the Brooklyn project in Eclipse (see [IDE Setup](#)).

Now create a debug configuration by clicking `Run | Debug configurations...`. You will then be presented with the Debug Configuration dialog.

Select `Remote Java Application` from the list and click the 'New' button to create a new configuration. Set the name to something suitable such as 'Remote debug on 8888'. The Project can be set to any of the Brooklyn projects, the Connection Type should be set to 'Standard (Socket Attach)'. The Host should be set to either localhost or 127.0.0.1 and the Port should be set to 8888. Click 'Debug' to start debugging.

IntelliJ Setup

To debug using IntelliJ, first open the Brooklyn project in IntelliJ (see [IDE Setup](#)).

Now create a debug configuration by clicking `Run | Edit Configurations`. You will then be presented with the Run/Debug Configurations dialog.

Click on the `+` button and select 'Remote' to create a new remote configuration. Set the name to something suitable such as 'Remote debug on 8888'. The first three sections simply give the command line arguments for starting the java process using different versions of java, however we have already done this in [Starting Brooklyn with a debug listener](#). The Transport option should be set to 'Socket', the Debugger Mode should be set to 'Attach', the Host should be set to localhost or 127.0.0.1 (or the address of the remote machine if you are not using an SSH tunnel), and the Port should be set to 8888. The 'Search sources' section should be set to `<whole project>`. Click OK to save the configuration, then select the configuration from the configurations drop-down and click the debug button to start debugging.

Testing

The easiest way to test that remote debugging has been setup correctly is to set a breakpoint and see if it is hit. An easy place to start is to create a breakpoint in the `ServerResource.java` class, in the `getStatus()` method.