Monterey User Guide

4.0.0-SNAPSHOT

User Guide

Cloudsoft Corporation **2012-01-23**

Contents

1. **Contents** i
2. **Introduction** 1
3. **Actor Developer Guide** 3
4. **Deploying and Managing a Monterey System** 10
5. **Appendix** 19
6. **Brooklyn Quick-Start Guide** 20
7. **Defining Applications with Brooklyn** 21
8. **Understanding Management and Writing Policies** 28
9. **Writing Custom Entities** 35
10. **Brooklyn Extras** 37
11. **Cloudsoft Developer License** 39

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

1 Introduction

1.1 Welcome to Monterey!

Monterey is a framework intended to simplify the development of scalable, distributed actor-based systems. Written in Java, it allows actors to be developed in any JVM language.

In this document you will learn how to develop actors and considerations to make when writing them, then how to deploy them in a Monterey network.

1.1.1 Key Concepts

Monterey is an implementation of the Actor Model, where a system is treated as a collection of concurrent independent **actors** running at **venues**, communicating with each other asynchronously to achieve the system's overall goals.

In addition to simplifying actor creation and management, Monterey also provides **intelligent application mobility**, allowing these actors to be moved without disruption to improve efficiency, _e.g._ load balancing or reducing wide-area latency.

Actors

Actors perform specific local tasks for the system, calling to other actors as needed. The Monterey framework currently supports the following mechanisms of communication:

- **Point-to-point communication** (private messaging): each actor can directly address other actors by their ID, where they know this ID through design or a previous communication
- **Publish-subscribe communication** (topic messaging): each actor can publish to a topic, to which multiple actors can subscribe, where topics are defined by design

These mechanisms are exposed to actors through an `ActorContext`, giving full access to the Monterey framework.

Monterey can also provide shim classes to make actor development even simpler in certain contexts, such as remotable classes, CDI beans, or Akka code. (Some of these shims are still in development, as are other messaging schemes for use cases including shared queues with round-robin or nearest available neighbour delivery.)

Venues

Monterey hosts a system's actors at one or more **venue** containers which can be distributed over many physical machines and even multiple geographic sites. Each venue is responsible for:

- Managing each of its contained actors according to a well-defined lifecycle
- Facilitating communication among the actors hosted locally and in remote venues
- Providing JMX hooks to allow external management tools to interact with the venue and its actors.

Venues can be started as standalone OSGi container processes, or they can be embedded in other processes such as servlets or desktop applications.

Brokers

Actors in the Monterey framework communicate with each other through message brokers, with ActiveMQ and Qpid supported out of the box. By default, these brokers are provisioned and managed by the framework automatically, but it is possible to configure to use a certain type of broker or specific broker instances, and to configure where brokers are provisioned. For example, where communication is local to a single machine, brokers can be set up at the same machine (with a same-JVM optimized broker in development).

Mobility

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

Monterey's patented technology allows actors to be **migrated** between venues, and the brokers in use to be **switched**, without disrupting computation or communication among the actors. These dynamic application changes are extremely efficient, making it suitable for relatively frequent load-balancing as well as wide-area location optimization.

This application mobility is exposed through JMX, and can be used either manually or by automated policies which can perform load balancing, latency optimization, cost reduction, and many more functions. With mobility built in, Monterey makes it easy to build applications which can take full advantage of cloud's dynamic capabilities, solving the technical challenge of how to support runtime agility, and letting the developer quickly focus on the when and why of scaling and optimizing location.

1.1.2 Related Software

Brooklyn

Monterey comes bundled with the Brooklyn subsystem which acts as the Monterey Control Plane. This provides the ability to control Monterey through "infrastructure as code", allowing automated/autonomic deployment of the services supporting and supported by your application.

Brooklyn is developed and maintained by Cloudsoft, with example usage for Monterey included in this document.

jclouds

jclouds is an open source Java library that provides portable compute and storage abstractions for most major cloud service providers. It is used within Monterey to allow your application to be deployed across many different providers using a standardised API, whilst also allowing access to provider-specific APIs where needed.

jclouds is developed and maintined by an active OSS community at jclouds.org. Cloudsoft provides professional open source support and certification for jclouds. Example usage relevant to Monterey deploying to various cloud providers is included in this document.

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

2 Actor Developer Guide

2.1 Creating a Project

Writing actors using Monterey is straightforward. This section introduces actors, point-to-point and pub-sub messaging, and the implementation considerations to observe when writing actors.

2.1.1 Prerequisites

The instructions that follow require you have the Java 6 JDK, curl, wget and Maven 3 installed.

You will also need to set up ssh authorized keys so you can ssh to localhost without a password (there is a quick summary of how to do so in Appendix A).

2.1.2 Creating a new Monterey Actor Project using the Maven Archetype

There is a Maven archetype which will build a skeleton of a Monterey actor project for you:

```
% mvn archetype:generate
   -DarchetypeRepository=http://ccweb.cloudsoftcorp.com/maven/libs-release-local/
   -DarchetypeGroupId=monterey
   -DarchetypeArtifactId=monterey-actors-archetype
   -DarchetypeVersion=4.0.0-SNAPSHOT
```

This command selects the **Monterey Actors Archetype** project. The project requires you give the following properties:

- groupId: the Maven group you wish the project to belong to (e.g. `com.example.app`)
- artifactId: the name of the Maven project. This will be the name of the directory in which your project is generated.
- version: the version of your new Maven project (defaults to 1.0-SNAPSHOT)
- package: the package in which your actor class will be placed (defaults to your entry for groupId)
- actorClass: the name of the actor class to be generated

Once this has finished, check your new project builds correctly:

```
% cd my-application
% mvn clean package
```

2.1.3 Converting an Existing Project to a Monterey Actor Project

Your project will depend on Monterey's actor API. This is available from Cloudsoft's Maven repository. You should add the following XML fragments to your project's pom.xml:

```
<repositories>
  <repository>
    <id>cloudsoft-maven-repository</id>
    <url>http://developer.cloudsoftcorp.com/download/maven2/</url>
  </repository>
</repositories>
<dependencies>
  <dependency>
    <groupId>monterey</groupId>
    <artifactId>monterey-actor-api</artifactId>
    <version>4.0.0-SNAPSHOT</version>
  </dependency>
</dependencies>
```

To use the venue logging, you will also need:

```
<dependency>
  <groupId>monterey</groupId>
```

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

```
    <artifactId>monterey-venue-util</artifactId>
    <version>4.0.0-SNAPSHOT</version>
</dependency>
```

2.1.4 Actor Basics

Once you have the actor API available in your project, you can start writing code for actors.

2.1.5 Writing Actors

Actors must implement the `monterey.actor.Actor` interface. The simplest "do-nothing" actor looks as follows:

```
import monterey.actor.*;
public class HelloWorldActor implements Actor {
    public void init(ActorContext context) { }
    public void onMessage(Object payload, MessageContext messageContext) { }
}
```

The actor created by the Maven archetype also has `start`, `suspend` and `resume` methods to support the **Suspendable** interface, and a `terminate` method to support the **Terminable** interface.

To be able send messages (and do other useful things), an actor uses its instance of `ActorContext` passed in to the actor's init method. Caching this in an instance field permits our actor to respond to received messages in the body of its `onMessage` method. For example, a simple actor that replies "hello" to the sender of any message it receives is shown below:

```
import monterey.actor.*;
public class HelloWorldActor implements Actor {
    private ActorContext context;
    public void init(ActorContext context) {
        this.context = context;
    }
    public void onMessage(Object payload, MessageContext messageContext) {
        ActorRef sender = messageContext.getSource();
        context.sendTo(sender, "Hello, "+payload);
    }
}
```

Instructions for building deployable artifacts from these actors and launching them at new venue instances is described under Deploying and Managing a Monterey System The simplest way to experiment with actors and to write your unit tests is with the venue test harness.

2.1.6 Running Actors in the Venue Test Harness

The venue test harness can be used to run actors in a simulated environment. This allows tests to be written that programmatically create actors, communicate with actors, and interact with these actors.

A common development model is to unit-test the business logic in actor code where possible, and to use the venue test harness for unit-testing message handing of actors and interactions between multiple actors. Then to deploy the actors to a venue running in a public or private cloud.

To use the venue test harness, the following Maven dependency is required:

```
<dependency>
  <groupId>monterey</groupId>
  <artifactId>monterey-testharness</artifactId>
  <version>4.0.0-SNAPSHOT</version>
</dependency>
```

An example of using the test harness is shown below. It creates a new actor of type `my.package.MyActor`, and then sends a message directly to that actor:

```
public void testMyActor() throws Exception {
    VenueTestHarness harness = VenueTestHarness.Factory.newInstance();
```

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

```
        ActorRef actor = harness.newActor(new ActorSpec(my.package.MyActor.class.getCanonicalName(),
            "myDisplayName"));
        harness.sendTo(actor, "my message");
        harness.shutdown();
    }
```

Basic Setup

As illustrated above, the `newActor` method creates a new actor at the harness, given an `ActorSpec` pointing to the no-argument `Actor` class and given a particular name. Also as illustrated above, the `sendTo` method will send a message from the test harness to a specific actors. Analogously, new venues can be created with the `newVenue` method, and topic messages published using the `publish` method.

The harness also allows the test to listen to topic messages via the `subscribe` method. (However it is not currently possible within the harness to access private response to messages that the harness sends.)

Actor Migration

Actors can be migrated by using either of the methods:

```
migrateActor(ActorRef actor)
migrateActor(ActorRef actor, VenueId oldVenueId, VenueId newVenueId)
```

The first of these migrates the actor to a different randomly selected venue. The latter gives much finer grained control and error checking (requiring the actor to be at `oldVenueId`). The set of available venues can be discovered by calling `getVenueIds()` and the current venue of the actor by calling `getVenueOfActor(ActorRef)`.

Broker Configuration

The test harness starts its own JMS broker (by default, ActiveMQ) to facilitate actor-to-actor communication. By default the broker opens TCP port 61616 to accept incoming JMS connections but this can be overridden by specifying the desired port number in the `monterey.venue.jms.port` system property when the venue is launched.

The broker's type can be set by passing a String argument to the test harness' `newInstance` method. Valid values are "activemq", "qpid" and "qpid+plugin". The last of these runs with a custom plugin in the Qpid broker, to allow for much more efficient actor migrations.

2.1.7 Further Examples

Actor examples are available for download from GitHub. To retrieve the source, execute the following command:

```
git clone git@github.com:cloudsoft/monterey-v4-examples.git
```

You can also browse the code on the web: Monterey Examples on Github.

2.1.8 Implementation Considerations

Monterey strives to simplify the development of complex actor-based systems, and to this end provides an execution model that provides easily-understandable system behavior and reduces the amount of actor code that needs to be written. Following the guidelines below will maximise the benefits of Monterey.

Simple actors are simple to write

The behavior of a simple actor may be fully defined in the implementation of its `onMessage` method.

Actors execute concurrently

Actors are executed independently by the system, even within the same venue. An actor that blocks for long periods (see below) will not prevent the execution of the other actors in the system.

For a given actor, framework methods are invoked serially

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

None of the actor's message-handling or lifecycle methods will be invoked while another is being executed, which obviates the need for explicit synchronization in many cases. Although these semantics are similar to "single-threaded" execution models, Monterey makes no guarantees about the relationship between actors and underlying Java threads; such details are unspecified. To wit, venues are not required to call a given actor's methods in the same thread, nor are they required to dedicate a thread to each actor. The use of thread-local storage within an actor is therefore strongly discouraged.

2.1.9 Writing Advanced Actors

While Monterey exists to simplify actor based systems, the attraction of Java is the power available. Please be aware of the following semantics when writing more powerful actors.

Migrating actors with arbitrary internal state

For the actor to be migratable by the platform, it must be able to provide a `Serializable` representation of its state, and be able to restore said state entirely from such a representation (see the `Suspendable` interface for details).

Creating and managing additional threads

Actors are permitted to create additional JVM threads, but the actor then assumes responsibility for all aspects of their management, and for any necessary synchronization with framework-invoked methods. It is preferable to spawn new actors (with no shared state, that communicate asynchronously using messages) and allow the framework to manage their concurrency.

Implications of being Suspendable

Suspendable actors must ensure that no additional threads are still running when its suspend method returns, and that the additional threads are restored along with the rest of the actor's state before the resume method returns.

Blocking operations

Actors are not prevented from blocking the message-handler and lifecycle methods for extended periods by performing time-consuming operations in framework-invoked methods, but doing so may adversely affect performance, restrict when that actor can be migrated, and possibly prevent the containing venue from shutting down in a timely manner. Note also that since incoming messages are queued until they can be processed by the receiver, a slow consumer may eventually exhaust the physical resources of the host machine.

2.1.10 Actor Communication Patterns

Actors communicate with other actors (possibly in remote venues) through the exchange of messages to affect the global behaviour of the system. Two distinct messaging patterns are supported:

Direct (Point-to-Point) Communication

An actor can send messages directly to any actor for which it holds a valid `ActorRef` ("actor reference") by invoking the `sendTo(ActorRef destination, Object payload)` method on its `ActorContext`.

The `ActorRef` is an opaque reference to an actor in the system, possibly residing in a remote venue. An `ActorRef` can be obtained as follows:

- Within the body of the onMessage method, the `MessageContext.getSource` method will return a reference to the actor that sent the message.
- When an actor is created (by calling `ActorContext.createActor`), a corresponding `ActorRef` is returned.
- A reference to an existing actor can be obtained by invoking `ActorRef.lookupActor` with the unique ID of the desired actor.
- An actor can obtain a reference to itself via `ActorContext.getSelf`.
- `ActorRef` instances may be included in the payloads of messages and thus communicated between actors.

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

We expect to add more flexible actor-lookup capabilities in the near future.

Publish-Subscribe

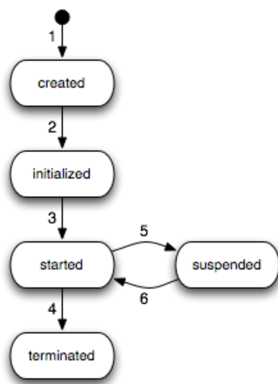
Actors can also publish and subscribe to named topics. Each message published on a given topic will be delivered to all actors that are currently subscribed to that topic.

An actor can subscribe to a topic by calling `ActorContext.subscribe` with the name of the desired topic, and unsubscribe by calling `ActorContext.unsubscribe`. To publish to a topic, an actor calls `ActorContext.publish` with the topic name and message payload.

An actor receives topic messages in the same way as direct messages: via a call to the `onMessage` method. If necessary it can determine how the message was sent via the `getTopic` method on the provided `MessageContext` (`getTopic` returns null if the message was sent point-to-point).

2.1.11 Actor Lifecycle

Monterey actors are managed according to a well-defined lifecycle, summarized below. An actor can define methods to be invoked when various lifecycle events occur. This can be done using annotations on methods or by implementing appropriate interfaces, as follows:



Lifecycle Event	Description	Annotation	Interface Method
Construction	The actor is instantiated by invoking the appropriate class's default constructor.	n/a	n/a
Initialization	Readies the actor for use, including providing a usable ActorContext instance.	n/a	`Initializable.init`
Activation	Notified immediately after the framework has started the actor, but before any messages have been delivered.	`@PostStart`	`Suspendable.start`

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

Termination	Indicates that the actor should perform any necessary cleanup in advance of being terminated; no further messages will be delivered.	`@PreTerminate`	`Terminable.terminate`
Suspension	Indicates that the actor is about to be temporarily paused, possibly so it can be migrated to another venue.	`@PreSuspend`	`Suspendable.suspend`
Resumption	Restores the state of a previously-suspended actor.	`@PostResume`	`Suspendable.resume`

Note that multiple methods can be annotated for any of the given lifecycle events, and that interface and annotated methods can be mixed. In such cases, annotated methods will be called before the equivalent interface methods, however the invocation order of annotated methods is unspecified.

2.1.12 Implementing Suspend and Resume

Below is an example actor which implements the `Suspendable` interface (alternatively it could have used annotations). The example can also be found in the `Monterey Examples GitHub repository < https://github.com/cloudsoft/monterey-v4-examples>`.

The actor's state is a simple counter that is incremented on receipt of each message. The actor preserves this state when being suspended and resumed (including when being moved to a different location in between). The subscription done in `start()` will be automatically re-subscribed on resume.

```
import java.io.Serializable;
import monterey.actor.Actor;
import monterey.actor.ActorContext;
import monterey.actor.MessageContext;
import monterey.actor.trait.Suspendable;
public class SuspendResumeActor implements Actor, Suspendable {
    private ActorContext context;
    private long count;
    @Override
    public void init(ActorContext context) {
        this.context = context;
    }
    @Override
    public void onMessage(Object payload, MessageContext messageContext) {
        count++;
        // Publish the latest count (primarily for the purpose of testing this actor)
        context.publish("count", count);
    }
    @Override
    public void start(Object state) {
        count = 0;
        context.subscribe("topic1");
    }
    @Override
    public Serializable suspend() {
        return count;
    }
    @Override
    public void resume(Object state) {
        count = (Long) state;
```

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

```
        }
    }
```

2.1.13 Logging

For the Monterey 4.0 GA, logging will be configurable so you can choose your favorite logging framework (e.g SLF4J, Log4J, etc). The milestone versions use plain `java.util.logging`. See http://download.oracle.com/javase/1.4.2/docs/guide/util/logging/overview.html for details.

Developers may use any logging framework when implementing actors. It is also possible to use the same logging as the Monterey venues so that user-logging is interleaved with that for Monterey framework events (which has the advantage of making the ordering of framework/user log messages explicit).

The Monterey logging classes reside in the `monterey.logging` package. The `LoggerFactory` is used to obtain a `Logger` instance, which can then be used to log at any of five named levels: trace, debug, info, warn and error. Note that the `LoggerFactory` is subject to change in future milestone releases; the `Logger` interface is more stable and is the same as that used by jclouds.

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

3 Deploying and Managing a Monterey System

3.1 Installation

Once the system's actors have been written, the application can be started and managed using Cloudsoft's Brooklyn or other management tools.

3.1.1 Prerequisites

The instructions that follow require you have the Java 6 JDK, curl, wget and Maven 3 installed.

You will also need to set up ssh authorized keys so you can ssh to localhost without a password to run the examples in the localhost location (there is a quick summary of how to do so in Appendix A).

3.1.2 Running Monterey

The recommended way to run Monterey is to use Brooklyn. Brooklyn can manage the deployment of Monterey venues and actors and control the system when it is running. It can also manage other services.

Monterey can also be run standalone by manually starting venues. The system can then be controlled with JMX.

3.1.3 Deploying the System with Cloudsoft Brooklyn

3.1.4.1 Creating a Monterey Application from the Maven Archetype

There is a Maven archetype which will build a skeleton of a Monterey Application project for you:

```
% mvn archetype:generate
   -DarchetypeRepository=http://ccweb.cloudsoftcorp.com/maven/libs-release-local/
   -DarchetypeGroupId=monterey
   -DarchetypeArtifactId=monterey-brooklyn-archetype
   -DarchetypeVersion=4.0.0-SNAPSHOT
```

This command selects the **Monterey Application Archetype**. You will need to provide:

- groupId: the Maven group you wish the project to belong to (e.g. `com.example.app`)
- artifactId: the name of the Maven project. This will be the name of the directory in which your project is generated.
- version: the version of your new Maven project (defaults to 1.0-SNAPSHOT)
- package: the package in which your application class will be placed (defaults to your entry for groupId)
- actorClass: the **fully qualified** class name of an actor in a Monterey actor project
- actorProjectArtifactId: the artifactId of the actor project
- actorProjectGroupId: the groupId of the actor project containing your actor code
- actorProjectVersion: the version of the actor project
- applicationClass: the name of your new application script

Once this has finished, check your new project builds correctly:

```
% cd my-application
% mvn clean package
```

The script in the generated project should look something like the following:

```
package test;
import brooklyn.entity.basic.AbstractApplication
import brooklyn.launcher.BrooklynLauncher
import brooklyn.location.basic.LocalhostMachineProvisioningLocation
import brooklyn.entity.messaging.activemq.ActiveMQBroker
import monterey.brooklyn.MontereyConfig
```

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

```
/**
 * Monterey Application script - generated from Archetype
 */
public class TestApplication extends AbstractApplication {
    public static void main(String[] argv) {
        // Create the app, configure it and have Brooklyn manage it
        TestApplication app = new TestApplication(displayName: "TestApplication app")
        def config = new MontereyConfig()
        config.network(this, name: "TestApplication Network") {
            bundles {
                url "wrap:mvn:monterey-v4-examples/simple-actors/4.0.0-M1"
            }
            actors(defaultStrategy: "pojo") {
                start "ExampleActorName", type: "monterey.example.ExampleActor"
            }
        }
        BrooklynLauncher.manage(app)
        // Start the app on localhost
        LocalhostMachineProvisioningLocation loc = new LocalhostMachineProvisioningLocation(count:10)
        app.start([loc])
    }
}
```

In this example the main method defines the application and passes it to `BrooklynLauncher` to be managed. It is then started in a localhost location - any other location could be used, such as EC2 or GoGrid. The Monterey network is defined in the `init()` method. It uses an ActiveMQ broker, deploys a monterey-example-actors bundle to a single venue and indicates that EchoActor should start in the venue.

You can take this archetype example script and modify it. To run your application, you should alter:

- the bundles URL to refer to your actors code
- the actors block to list the actors you wish to instantiate.
- each actor's type to reference the type of actor you have defined

Launching the Application

To build the projects run `mvn clean install` in the root folder. The maven-shade plugin builds everything that can be included on our classpath into a single jar.

Run your `TestApplication` class: `java -cp target/test-app-0.0.1-SNAPSHOT.jar test.TestApplication`.

The Brooklyn launcher includes a web-console which we can connect to at http://localhost:8081 with username 'admin' and password 'password'. The Brooklyn web-console can be used to monitor and manage the Monterey network.

Running in the Cloud

With a small modification to the Brooklyn launch code we can run the application in the cloud. Change the main method of `TestApplication` to run it on AWS-EC2, using your AWS credentials and SSH keys rather than the placeholders:

```
public static void main(String[] argv) {
    // Create and configure the app as before..
    // Start the app on AWS-EC2
    JcloudsLocationFactory locFactory = new JcloudsLocationFactory([
        provider : "aws-ec2",
        identity : "12345678901234567890",
        credential : "09876543210987654321O/09876543+210987654",
        sshPrivateKey : new File("/home/bob/.ssh/id_rsa.private"),
        sshPublicKey : new File("/home/bob/.ssh/id_rsa.pub")
    ])
    JcloudsLocation loc = locFactory.newLocation("eu-west-1")
    loc.setTagMapping([
        (Venue.class.getName()):[
            securityGroups:["brooklyn-all"]],
        (ActiveMQBroker.class.getName()):[
            securityGroups:["brooklyn-all"]]
```

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

```
        ])
        app.start([loc])
    }
```

This uses jclouds to create a location, supplying the AWS-EC2 credentials and the RSA key to use for logging into the VMs. The 'tag mapping' allows customisation of the settings for particular types of entity. In this example the security group for venue entities and for ActiveMQBroker entities is set to 'brooklyn-all', which opens all ports to provide access over JMX. In a future release it will be possible to lock down the ports to use, and auto-configure the security group.

The application can be run in the same way the localhost version. It will create a VM for ActiveMQ and a VM for a Monterey venue.

3.1.4.2 Creating a Monterey Application Manually

A number of dependencies are required to build a Monterey Application. In Maven these are:

```
<dependency>
    <groupId>monterey</groupId>
    <artifactId>monterey-brooklyn-integration</artifactId>
    <version>${project.version}</version>
</dependency>
<dependency>
    <groupId>brooklyn</groupId>
    <artifactId>brooklyn-launcher</artifactId>
    <version>${brooklyn.version}</version>
</dependency>
<dependency>
    <groupId>org.osgi</groupId>
    <artifactId>org.osgi.enterprise</artifactId>
    <version>4.2.0</version>
</dependency>
```

Maven, or your chosen build system, must be told to use a Groovy compiler. For example, to use the Eclipse Groovy compiler in a Maven project:

```
<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
        <compilerId>groovy-eclipse-compiler</compilerId>
        <fork>true</fork>
        <verbose>false</verbose>
        <source>1.6</source>
        <target>1.6</target>
    </configuration>
    <dependencies>
        <dependency>
            <groupId>org.codehaus.groovy</groupId>
            <artifactId>groovy-eclipse-compiler</artifactId>
            <version>2.5.1-M3</version>
        </dependency>
        <dependency>
            <groupId>org.codehaus.groovy</groupId>
            <artifactId>groovy-eclipse-batch</artifactId>
            <version>1.8.0-03</version>
        </dependency>
    </dependencies>
</plugin>
```

3.1.4.3 Specifying Monterey Networks

Brooklyn is the standard way to deploy, run and manage a Monterey network. Brooklyn provides a Groovy DSL for specifying details of your desired network along with the application to deploy, and allows any aspect of this to be changed at runtime.

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

The Brooklyn integration is being enhanced for future milestones to improve changing the Monterey network at runtime and to support more policies.

Networks are built by creating an instance of MontereyConfig and building the configuration using a custom DSL. The DSL uses Groovy syntax to build the Monterey application and can have Groovy code intermixed with Monterey component configuration statements.

The starting point for the configuration is a network instance:

```
def monterey = config.network(owningApplication, displayName:"Demo Network") {
    ...
}
```

The configuration must be owned by an Application and must specify brokers, bundles and actors. The number of brokers and venues to start initially in each location defaults to one, but can also be given as parameters `initialNumBrokersPerLocation` and `initialNumVenuesPerLocation`.

3.1.4.4 Brokers

The brokers keyword is used to indicate the type of broker to manage inter-actor communication. Broker-specific configuration can be given after its type. For example to start an ActiveMQ broker that uses port 5678 for JMX operations:

```
brokers(ActiveMqBroker, jmxPort:5678)
```

Additional properties supported are openWirePort for ActiveMQ and amqpPort for Qpid.

Monterey currently supports Qpid (QpidBroker and QpidMontereyBroker) and ActiveMQ (ActiveMq) brokers. External brokers that are not managed by Monterey can be specified by giving the URL to connect to and setting the `initialNumBrokersPerLocation` property of the network to 0, as follows:

```
config.network(..., initialNumBrokersPerLocation:0) {
    brokers(QpidBroker) {
        broker "amqp://user:password@clientid/virtualhost?brokerlist='tcp://
qpid1.example.org:5672'"
        broker "amqp://user:password@clientid/virtualhost?brokerlist='tcp://
qpid2.example.org:5672'"
    }
...
```

If no brokers are specified Monterey will use ActiveMQ. To use the Qpid broker with Monterey specific customisations, specify QpidMontereyBroker as the type, and set the actorMigrationMode property of the network to USE_BROKER_WITH_ATOMIC_SUBSCRIBER_SWITCH as follows:

```
config.network(..., actorMigrationMode:ActorMigrationMode.USE_BROKER_WITH_ATOMIC_SUBSCRIBER_SWITCH) {
    brokers(QpidMontereyBroker, amqpPort:15671)
...
```

3.1.4.5 Bundles

Bundles to be deployed are specified in the Pax URL format. See http://team.ops4j.org/wiki/display/paxurl/Documentation for information on specific URL formats.

For example, to deploy a monterey-example-actors jar from a Maven repository and commons-logging-1.1 from the working directory:

```
bundles {
    url "wrap:mvn:monterey-repo/monterey-example-actors/0.0.1-SNAPSHOT"
    url "wrap:file:commons-logging-1.1.jar"
}
```

3.1.4.6 Actors

Actors to be run in the network are declared in an actors block, with the form `name type:className, config`, where name is used for display and className corresponds to a class in a given bundle. The default strategy

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

to use when creating actors should be indicated at the head of the block and actor-specific configuration parameters should be given in a config map:

```
actors {
    start "pong", type: "example.pingpong.PongActor"
    start "ping", type: "example.pingpong.PingActor", config: [logLevel: FINEST]
}
```

If actors of a particular type share configuration you can simplify your script by writing:

```
actors {
    type "example.Worker", config: [ iterations: 5000 ]
    start "worker1", type: "example.Worker"
    start "worker2", type: "example.Worker"
    start "worker3", type: "example.Worker", config: [ iterations: 10000 ]
}
```

Workers one to three will inherit the config map from the type. Worker three will overwrite the iterations key with its own setting.

Runtime environment and Java options may be specified in environment and javaOptions blocks. Parameters should be given as a comma-separated list of key: value pairs.

For example, to start the PingActor and PongActor with property example.pingpong.verbose set to true:

```
actors {
    ...
    start "pong", type: "example.pingpong.PongActor", description: "The Pong Actor"
    start "ping", type: "example.pingpong.PingActor", description: "The Ping Actor"
}
javaOptions(
    "example.pingpong.verbose": true
)
```

In this milestone release Monterey only supports providing -D properties to programs; there is no way set the heap size with -Xmx, for example. This will supported in a later release.

3.1.5 Managing the System with Cloudsoft Brooklyn

The Monterey entities in brooklyn expose effectors and sensors for managing and monitoring the application. These can be used manually, or by policies.

An actor has the following effectors:

- terminate(): terminates this actor
- migrate(VenueId oldVenueId, VenueId newVenueId): migrates this actor from the old venue to the new venue

An actor has the following sensors:

- venue id: the venue currently hosting the actor
- actor status: the status of the actor, such as RUNNING; this is a string version of the enumeration `monterey.venue.management.ActorStatus`
- total messages received: total number of messages that have been delivered to this actor; never reset
- total messages sent: total number of messages that have been sent by this actor; never reset
- up time: total time (millis) that actor has been non-paused
- processing time: total time (millis) that actor has spent processing requests

A venue has the following sensors:

- num actors: number of live actors in the venue
- total messages received:total number of messages that have been delivered to actors in this venue
- total messages sent: total number of messages that have been sent by actors in this venue

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

For examples, a policy could monitor the workload of each actor, to balance them across a set of venues. The policy could use an `Enricher` to calculate the workload (e.g. average message-rate over the past 10 seconds), and could call the `actor.migrate` effector to move actors to the least loaded venue.

Note: in a future milestone release additional effectors will be added, such as `Venue.terminate()`. There will also be additional sensors such as average message-rates over the last several seconds.

3.1.6 Launching a POJO Venue

The recommended way to launch a Monterey network, including venues, is to use Brooklyn. However, a standalone (POJO) Monterey venue can be manually launched from the command-line as follows:

```
   java \
     -cp target/monterey-venue-pojo-4.0.0-SNAPSHOT-with-dependencies.jar:target/test-
actors.jar \
     -Dcom.sun.management.jmxremote \
     -Dcom.sun.management.jmxremote.port=37585 \
     -Dcom.sun.management.jmxremote.ssl=false \
     -Dcom.sun.management.jmxremote.authenticate=false \
     monterey.venue.pojo.Main \
     --venueId=Chicago-44 \
     --brokerType=activemq \
     --connectionUrl=tcp://192.168.7.44:61616/
```

The `-cp` option configures the JVM's classpath to include the Monterey venue code and the classes for any actors that will be instantiated in the venue.

The four `-D` parameters define Java system properties to enable and configure remote management over JMX, as described at http://tomcat.apache.org/tomcat-5.5-doc/monitoring.html.

`monterey.venue.pojo.Main` is the program entry point.

The remaining parameters configure the new venue:

- `--brokerPort` Required. The JMS broker port number
- `--connectionUrl` Required. A comma separated list of the JMS broker's connection URLs.
- `--venueId` Optional. Assigns it a globally unique identifier. Behind the scenes, Monterey venues use JMS to facilitate actor communication. A venue can use an existing JMS broker, or it can start one of its own. If no id is assigned, one will be randomly generated.
- `--brokerType` Optional. Identifies the JMS implementation in use, and must be one of activemq (the default) or qpid.
- `--jmxPort` Optional. The broker JMX port number
- `--brokerConfig` Optional. The broker configuration e.g internal or external. If internal is specified, the venue will start its own ActiveMQ broker (on the specified connectionUrl)

3.1.7 Venue Management Hooks (JMX)

Monterey venues are managed through JMX, using the platform's MBeanServer. JConsole and other JMX-based monitoring and management software can connect to the venue and obtain information about the venue and the actors it contains.

There is an MBean for each venue. This has an object name of the form `monterey:type=Venue,id=XXXX`, where `XXXX` is the venue's unique identifier.

There is an MBean for each actor (in the JVM of the venue it is located in). This has an object name of the form `monterey:type=Actor,venue=XXXX,id=YYYY`, where `XXXX` is the unique ID of the venue containing the actor and `YYYY` is a the unique ID of the actor.

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

3.1.8 Debugging Monterey Venues

When writing applications using actors it is helpful to be able to check the status of these components. The following sections describe some useful techniques. Brooklyn starts venues as Karaf containers, and uses ActiveMQ as a JMS message broker. Check that local ActiveMQ and venue instances have been shutdown, before running subsequent tests, using the following commands:

```
$ ps aux | grep activemq
$ ps aux | grep karaf
$ lsof -i TCP:<port>
```

3.1.9.1 JConsole

When running the venue locally, it is possible to use JConsole to connect to the running Java process and inspect attributes of the venue.

3.1.9.2 Logging

Brooklyn downloads the distribution packages and installers for entities to /tmp/brooklyn/installs. It creates run directories for entity instances in /tmp/brooklyn/apps/<uuid>/entities/<entity>-<uuid>, using the uuid of the application as the directory name and the uuid of the entity itself as a sub-directory.

The log file for the venue can be found at /tmp/brooklyn/apps/<uuid>/entities/venue-<uuid>/data/log/venue.log.

3.1.9.3 Brooklyn Console

The Brooklyn web console is available on http://localhost:8081/detail/ and shows all deployed entities, their sensors and attribute values. This is useful to determine the host or port number to use to connect.

3.1.9.4 Karaf Web Console

Karaf provides a web based admin console, which allows checking which OSGi bundles are loaded, and whether dependencies are present. It is accessed at http://localhost:8181/system/console using the username 'admin' and password 'admin'.

3.1.9.5 ActiveMQ Web Console

This allows you to see what topics exist and how many messages have been enqued and dequeued on them, as well as listing all active subscriptions and connections. Connect to http://localhost:8161/admin/ using the user 'admin' and password 'activemq'.

3.1.9.6 Monterey Tools

Monterey includes a message monitoring tool. This connects to a broker and monitors messages sent to some or all `Actors` or Monterey topics. A script is provided to simplify operation, which can be run as follows:

```
$ ./monitor.sh actors [options] [actor1 actor2 ...]
$ ./monitor.sh topics [options] [topic1 topic2 ...]
```

If no topic or actor ids are supplied every topic or actor will be monitored. The output produced will look something like the following:

```
ElJBK0aa -> actor:roxosa0M - "AAPL,40463,902,1317594243188"
ytRu4LGH -> actor:roxosa0M - "IBM,18409,879,1317594247529"
E7klovJo -> actor:roxosa0M - "MSFT,2537,-181,1317594248790"
HJ3vsoTF -> actor:roxosa0M - "GOOG,52551,-1972,1317594250549"
```

Alternatively, the command can be run as follows, with options as described below:

```
$ java -jar monterey-tools-4.0.0-SNAPSHOT-with-dependencies.jar [options]
```

Option	Provided Value	Description

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

`-?, -h, --help`	-	Help
`-b, --brokerType`	<Type of broker: 'activemq' or 'qpid'>	Broker type (default: activemq)
`-m, --monitor`	<Either: 'actors' or 'topics'>	Destination type to monitor
`-u, --connectionUrl`	<URL to connect to the broker>	Broker connection URL (default: tcp://localhost:61616/)

3.1.9.7 Karaf SSH Console

Karaf also provides a command-line interface accessible over SSH. This gives a very comprehensive set of commands for examining and manipulating the OSGi environment. Additionally, there are custom Monterey venue commands, such as `venue: list` which shows all the actors deployed to the venue. Other commands that may prove useful are `log:display` and `osgi:list`. Tab completion and online help are available, as well as command history and other shell features like pipes and grep.

Detailed information on Karaf commands can be found in the Karaf documentation.

Connect to Karaf with username 'admin' and password 'admin'.

```
   $ ssh -o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null admin@localhost -p 8101
   Warning: Permanently added '[localhost]:8101' (DSA) to the list of known hosts.
   admin@localhost's password:
    __  __             _
   |  \/  | ___  _ __ | |_ ___ _ __ ___ _   _
   | |\/| |/ _ \| '_ \| __/ _ \ '__/ _ \ | | |
   | |  | | (_) | | | | ||  __/ |  |  __/ |_| |
   |_|  |_|\___/|_| |_|\__\___|_|   \___|\__, |
                                         |___/
   Monterey Venue (4.0.0-SNAPSHOT)
   Copyright 2009-2011 by Corporation Ltd. All Rights Reserved
   Supplied under license http://www.cloudsoftcorp.com/license/montereyDeveloperEdition
   or such subsequent license agreed between Cloudsoft Corporation Ltd and the licensee.
   monterey>
   monterey>  venue:list
   ID         NAME                 STATUS     TYPE
DESCRIPTION
   ---------- -------------------- ---------- ------------------------------
----------------------------------------
   waIQKvc8   ping                 RUNNING    example.pingpong.PingPongActor
Ping Actor
   yCFTbemb   pong                 RUNNING    example.pingpong.PingPongActor
Pong Actor
   monterey>
   monterey>  osgi:info 71
   Monterey Venue OSGi (71)
   ------------------------
   DESCRIPTION
     Monterey 4.0.0 Venue
   LICENSE
     Copyright 2009-2011 by Cloudsoft Corporation Ltd. All rights reserved.
     Supplied under license http://www.cloudsoftcorp.com/license/montereyDeveloperEdition
     or such subsequent license agreed between Cloudsoft Corporation Ltd and the licensee.
   SEE ALSO
     http://developers.cloudsoftcorp.com/
   monterey>
   monterey>  packages:exports | grep monterey
       71 monterey.venue.osgi; version=0.0.0
       72 monterey.venue.command; version=0.0.0
       73 monterey.venue; version=0.0.0
       73 monterey.venue.jms; version=0.0.0
       73 monterey.controller; version=0.0.0
       73 monterey.actor.factory; version=0.0.0
```

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

```
    73 monterey.actor.factory.pojo; version=0.0.0
    74 monterey.logging; version=0.0.0
    74 monterey.util; version=0.0.0
    75 monterey.venue.management; version=0.0.0
    76 monterey.venue.jms.spi; version=0.0.0
    77 monterey.actor; version=0.0.0
    77 monterey.actor.annotation; version=0.0.0
    77 monterey.actor.trait; version=0.0.0
    96 monterey.venue.jms.activemq; version=0.0.0
monterey>
monterey>  osgi:list | grep Actor
[  77] [Active     ] [            ] [   60] Monterey Actor API (4.0.0.SNAPSHOT)
[  97] [Resolved   ] [            ] [   60] Monterey Actor Examples (0.1.0.SNAPSHOT)
monterey>
monterey>  ^D
Connection to localhost closed.
```

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

4 Appendix

4.1 Appendix A: Setting up ssh keys

Once ssh keys are set up correctly, you should be able to run the following without being prompted for your password:

% ssh localhost

- You'll need sshd to be running, with many linux distros this will aready be the case.
    - in Mac OS X switch on *Remote Login* in the *Sharing* pane of *System Preferences*
- Adjust your keys to allow password-less login, by running the following in a terminal window:
    - % ssh-keygen -t rsa
    - % chmod 600 ~/.ssh/id_rsa
    - % cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

5 Brooklyn Quick-Start Guide

5.1 Brooklyn Introduction

Brooklyn is a provisioning and management framework for big, distributed applications. At a glance, it lets you:

- Describe an application topology once, and use this definition to set it up (provision) and keep it up (management)
- Describe the application topology in code, for re-use, version control, power and readability
- Run multiple tiers and even varying stacks, configured and managed together
- Run in multiple locations with efficient, secure wide-area management

5.1.1 Prerequisites

This guide requires you have the Java 6 JDK, curl, wget and Maven 3 installed.

If you are using Eclipse, you will require the Groovy and Maven plugins.

- In Eclipse goto: Help -> Install New Software, or obtain the following using the Eclipse Marketplace.
- Groovy Plugin: http://dist.springsource.org/release/GRECLIPSE/e3.7
- Maven Plugin: http://download.eclipse.org/technology/m2e/releases

5.1.2 WorldwideSpringTravel Example

This is what a relatively complex sample application looks like:

```
public class WorldwideSpringTravel extends AbstractApplication {
        // define our multi-location tiers (using off-the-shelf entities)
        def tomcat = new TomcatWithDnsFabric(this,
            name:'SpringTravelWebApp', war:'spring-travel.war',
            domain:'www.travel.mycompany.com', dns:"ns.mycompany.com")
        def gemfire = new GemfireFabric(this, name:'SpringTravelGemfire')
        def monterey = new MontereyFabric(this,
            name:'SpringTravelBooking', osgi:'com.cloudsoft.spring.booking.impl')
        { //wire the tiers together
            tomcat.webCluster.template.setConfig(JavaEntity.JVM_PROPERTY("monterey.urls"),
                attributeWhenReady(monterey, Monterey.MGMT_PLANE_URLS) )
            monterey.setConfig(JavaEntity.JVM_PROPERTY("monterey.urls"),
                attributeWhenReady(gemfire, Gemfire.URLS) )
            monterey.policy << new MontereyLatencyOptimisationPolicy()
        }
}
```

This consists of a Tomcat web-app tier (which will automatically configure the DNS records for the given domain), a Gemfire datacache, and a Monterey processing fabric with a latency optimisation policy. The following command will start the application running across three locations:

```
new WorldwideSpringTravel().withConfig(LOGIN_CREDENTIALS).start(
        location:[ new AmazonEurope(), new GoGridUsWest(),
            new VcloudLocation("http://privatecloud.mycompany.com:8080") ]
```

A management plane is launched with the application, running policies for load-balancing, scaling, and optimizing placement. Keeping a handle on the WorldwideSpringTravel instance allows programmatic monitoring, manual management, and policy change; the management plane can also be accessed through a command-line console, a web console, or a REST web API. The management web console, shown below, shows the hierarchy of entities active in real-time--down to the level of each Tomcat process on a VM and every Monterey code segment, if desired.

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

6 Defining Applications with Brooklyn

6.1 Introduction

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.

6.1.1 The Basic Concepts - Entities and How to Use Them

At the heart of Brooklyn is the concept of an **entity**. Some entities correspond to actual machines or processes such as a `JBossNode` or a `RabbitMqNode`, others correspond to a group of other entities, such as a `TomcatCluster` or `MultiLocationMySqlTier`. Entities can be grouped or composed to form other entities, and are code so they can be extended, overridden, and modified. Their main responsibilities are:

- Provisioning the entity in the given location of 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

6.1.2 Entities, Application, Ownership and Membership

All entities have an owner entity, which creates and manages it, with two exceptions. Top-level `Application` entities are created and managed externally, perhaps by a script. **Templates** are the other exception, and are covered in more detail later.

A **Group** is an entity to which other entities are memembers. Membership can be used for whatever purposes the application definer wishes. A typical use-case is 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 **owners** (e.g. a multi-tier stack within a location).

6.1.3 Lifecycle and ManagementContext

An `Application Entity` defines the `ManagementContext` instance and is responsible for starting the deployment of the entire entity tree under its ownership. Only `Application Entity` can define the `ManagementContext`.

The management context entity forms part of the management plane. The management plane is responsible for the distribution of the `Entity` instances across multiple machines and multiple locations, tracking the transfer of events (subscriptions) between `Entity` instances, and the execution of tasks (often initiated by management policies).

An `Application Entity` provides a `start()` method which begins provisioning the management plane and distributing the management of entities owned by the application (and their entities, recursively). In a multi-location deployment, management operates in all regions, with Brooklyn entity instances being mastered in the relevant region.

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 visible in the management console.

Customizing 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.

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

For new entities joining an existing network, the entity is deployed to the management plane when it is wired in to an application i.e. by giving it an owner. Templates for new entities are also deployed to the management plane in the same manner.

6.1.4 Configuration, Sensors and Effectors

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 will make it available in all entities underneath unless it is overridden.

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

Configuration values can be specified in a configuration file (`~/.brooklyn/brooklyn.properties`) to apply universally, and programmatically to a specific entity and its descendants using the `entity.setConfig(KEY, VALUE)` method. Many common configuration parameters are available as "flags" which can be supplied in the entity's constructor of the form `new MyEntity(owner, config1: "value1", config2: "value2")`. Documentation of the flags available for individual constructors can be found in the javadocs, or by inspecting `@SetFromFlag` annotations on the `ConfigKey` definitions.

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 owner or other entities to track changes in an entity's activity. Effectors, can be invoked by an entity's owner 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!

Entities are Java classes and data can also be stored in internal fields. This data will not be inherited and will not be externally visible (and resilience is more limited), but the data will be moved when an entity's master location is changed. (See discussion of management below.)

6.1.5 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 a set of URLs to connect to a Monterey processing fabric (or a database tier or other entities) :

```
tomcat.webCluster.template.setConfig(JavaEntity.JVM_PROPERTY("monterey.urls"),
            attributeWhenReady(monterey, Monterey.MGMT_PLANE_URLS) )
```

The `attributeWhenReady(Entity, Sensor)` call causes the configuration value to be set when that given entity's attribue is ready. In the example, `attributeWhenReady()` causes the JVM system property `monterey.urls` to be set to the value of the `Monterey.MGMT_PLANE_URLS` sensor from `monterey` when that value is ready. As soon as a management plane URL is announced by the Monterey 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 is useful if you require three management plane URLs.

More information can be found in the javadoc for `DependentConfiguration`.

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

Note that `Entity.getConfig(KEY)` will block when it is used. Typically this does the right thing, blocking only when necessary without the developer having to think through explicit start-up phases, but it can take some getting used to.

You should be careful not to request config information until really necessary (or to use internal non-blocking "raw" mechanisms). Be ready in complicated situations to attend to circular dependencies. The management console gives sufficient information to understand what is happening and identify what is blocking.

6.1.6 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) for hosting the entity.

The idea is that you could specify the location as AWS and also supply an image id. You could configure the Tomcat entity accordingly: specify the path if the image already has Tomcat installed, or specify that Tomcat must be downloaded/installed. Entities typically use drivers (such as SSH-based) to install, start, and interact with their corresponding real-world instance.

6.1.7 Common Usage

6.1.8.1 Entity Class Hierarchy

By convention in Brooklyn the following words have a particular meaning, both as types (which extend Group, which extends Entity) and when used as words in other entities (such as TomcatFabric):

- Tier - anything which is homogeneous (has a template and type)
  - Cluster - in-location tier
  - Fabric - multi-location tier
- Stack - heterogeneous (mixed types of children)
- Application - user's entry point

- **template** entities are often used by groups to define how to instantiate themselves and scale-out. A template is an entity which does not have an owner and which is not an application.
- **traits** (mixins) providing certain capabilities, such as Resizable and Balanceable
- Resizable
- Balanceable / Moveable / MoveableWithCost

6.1.8.2 Off-the-Shelf Entities

Brooklyn includes a selection of entities already available for use in applications, including appropriate sensors and effectors, and in some cases include Cluster and Fabric variants. (These are also useful as templates for writing new entities.)

These include:

- Web: Tomcat, JBoss; nginx; GeoScaling; cluster and fabric
- Relational databases: MySQL, Derby
- NoSQL: Infinispan, Redis, GemFire
- Messaging: ActiveMQ, Qpid

For a full list see |chapter:systems available|.

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

6.1.8.3 Off-the-Shelf Policies

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

- Resizer Policy increases/decreases size of a Resizable entity based on an aggregate sensor value, the current size of the entity, and customized high/low watermarks.

  A Resizer 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.

6.1.8.4 Off-the-Shelf Enrichers

- Delta - converts absolute sensor values into a delta
- Time-weighted Delta - converts absolute sensor values into a delta/second
- Rolling Mean - converts the last N sensor values into a mean
- Rolling time-window mean - converts the last N seconds of sensor values into a weighted mean
- Custom Aggregating - 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.

6.1.8.5 Off-the-Shelf Locations

- SSH
- Compute: Amazon, GoGrid, vCloud, and many more (using jclouds)

  # use a special key when connecting to public clouds brooklyn.jclouds.private-key-file=~/.ssh/public_clouds/id_rsa brooklyn.jclouds.localhost.private-key-file=~/.ssh/id_rsa # need this one for localhost brooklyn.jclouds.aws-ec2.identity=ABCDEFGHIJKLMNOPQRST # AWS credentials brooklyn.jclouds.aws-ec2.credential=s3cr3tsq1rr3ls3cr3tsq1rr3ls3cr3tsq1rr3l brooklyn.geoscaling.username=cloudsoft # credentials for 'geoscaling' service brooklyn.geoscaling.password=xxx

These can also be set as environment variables (in the shell) or system properties (java command line). (There are also `BROOKLYN_JCLOUDS_PRIVATE_KEY_FILE` variants accepted.)

For any provider you will typically need to set `identity` and `credential` in the `brooklyn.jclouds.provider` namespace. Other fields may be available (from Brooklyn or jclouds).

`brooklyn.jclouds.public-key-file` can also be specied but can usually be omitted (it will be inferred by adding the suffix `.pub` to the private key). There should be no passphrases on the key files.

6.1.9 Examples

6.1.10.1 Integrating with a Maven project

If you have a Maven-based project, integrate this XML fragment with your pom.xml:

```
<dependencies>
    <dependency>
        <groupId>brooklyn</groupId>
        <artifactId>brooklyn-launcher</artifactId>
        <version>0.3.0-SNAPSHOT</version>
        <classifier>with-dependencies</classifier>
    </dependency>
</dependencies>
<repositories>
    <repository>
        <id>cloudsoft-maven-repository</id>
        <url>http://developer.cloudsoftcorp.com/download/maven2/</url>
    </repository>
```

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

```
</repositories>
```

6.1.10.2 Starting a Tomcat Server

The code below starts a Tomcat server on the local machine.

The main method defines the application, and passes it to the BrooklynLauncher to be managed. Here It is then started in a localhost location, but any location could be used including EC2 or GoGrid.

The init method declares the entities that comprise the app. In this case, it is a single tomcat instance.

The Tomcat's configuration indicates that the given WAR should be deployed to the Tomcat server when it is started.

```
class TomcatServerApp extends AbstractApplication {
    def tomcat = new TomcatServer(owner: this, httpPort: 8080, war: "/path/to/booking-
mvc.war")
    public static void main(String... args) {
        TomcatServerApp demo = new TomcatServerApp(displayName : "tomcat server example")
        BrooklynLauncher.manage(demo)
        demo.start([new LocalhostMachineProvisioningLocation(count: 1)])
    }
}
```

The code can be written in pure Java if preferred, using the long-hand syntax of `tomcat.setConfig(TomcatServer.HTTP_PORT, 80)` in lieu of the flags. The `wars` flag is also supported (with config keys `ROOT_WAR` and `NAMED_WARS` the long-hand syntax); they accept EARs and other common archives, and can be described as files or URLs, including a `classpath://org/acme/resources/xxx.war` syntax.

6.1.10.3 Starting a Tomcat Cluster in Amazon EC2

The code below starts a tomcat cluster in Amazon EC2.

In this milestone release, the following should be considered pseudo code:

```
class TomcatClusterApp extends AbstractApplication {
    DynamicWebAppCluster cluster = new DynamicWebAppCluster(
        owner : this,
        initialSize: 2,
        newEntity: { properties -> new TomcatServer(properties) },
        httpPort: 8080,
        war: "/path/to/booking-mvc.war")
    public static void main(String[] argv) {
        TomcatClusterApp demo = new TomcatClusterApp(displayName : "tomcat cluster example")
        BrooklynLauncher.manage(demo)
        JcloudsLocationFactory locFactory = new JcloudsLocationFactory([
                    provider : "aws-ec2",
                    identity : "xxxxxxxxxxxxxxxxxxxxxxxxxxx",
                    credential : "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",
                    sshPrivateKey : new File("/home/bob/.ssh/id_rsa.private"),
                    sshPublicKey : new File("/home/bob/.ssh/id_rsa.pub"),
                    securityGroups:["my-security-group"]
                ])
        JcloudsLocation loc = locFactory.newLocation("us-west-1")
        demo.start([loc])
    }
}
```

The `newEntity` flag in the cluster constructor indicates how new entities should be created. The WAR configuration set on the cluster is inherited by each of the TomcatServer contained (i.e. "owned") by the cluster.

The `DynamicWebAppCluster` is dynamic in that it supports resizing the cluster, adding and removing servers, as managed either manually or by policies embedded in the entity.

The main method creates a `JcloudsLocationFactory` with appropriate credentials for the AWS account, along with the RSA key to used for subsequently logging into the VM. It also specifies the relevant security group

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

which should enable the 8080 port configured above. Finally, a JcloudsLocation allows to select the Amazon region the cluster will run in.

6.1.10.4 Starting a Tomcat Cluster with Nginx

The code below starts a Tomcat cluster along with an Nginx instance, where each Tomcat server in the cluster is registered with the Nginx instance.

```
class TomcatClusterWithNginxApp extends AbstractApplication {
    NginxController nginxController = new NginxController(
        domain : "brooklyn.geopaas.org",
        port : 8000,
        portNumberSensor : Attributes.HTTP_PORT)
    ControlledDynamicWebAppCluster cluster = new ControlledDynamicWebAppCluster(
        owner : this,
        controller : nginxController,
        webServerFactory : { properties -> new TomcatServer(properties) },
        initialSize: 2,
        httpPort: 8080, war: "/path/to/booking-mvc.war")
    public static void main(String[] argv) {
        TomcatClusterWithNginxApp demo = new TomcatClusterWithNginxApp(displayName : "tomcat cluster with 
        BrooklynLauncher.manage(demo)
        JcloudsLocationFactory locFactory = new JcloudsLocationFactory([
                    provider : "aws-ec2",
                    identity : "xxxxxxxxxxxxxxxxxxxxxxxxxxx",
                    credential : "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",
                    sshPrivateKey : new File("/home/bob/.ssh/id_rsa.private"),
                    sshPublicKey : new File("/home/bob/.ssh/id_rsa.pub"),
                    securityGroups:["my-security-group"]
                ])
        JcloudsLocation loc = locFactory.newLocation("us-west-1")
        demo.start([loc])
    }
}
```

This creates a cluster that of Tomcat servers, along with an Nginx instance. The `NginxController` instance is notified whenever a member of the cluster joins or leaves; the entity is configured to look at the `HTTP_PORT` attribute of that instance so that the Nginx configuration can be updated with the ip:port of the cluster member.

The beauty of OO programming, of course, is that classes can be re-used. The compound entity we've created above is available off-the-shelf as the `LoadBalancedWebCluster` entity, as used in the following example.

6.1.11 Starting a Multi-location Tomcat Fabric

```
class TomcatFabricApp extends AbstractApplication {
    Closure webClusterFactory = { Map flags, Entity owner ->
        Map clusterFlags = flags + [newEntity: { properties -
> new TomcatServer(properties) }]
        return new DynamicWebAppCluster(clusterFlags, owner)
    }
    DynamicFabric fabric = new DynamicFabric(
            owner : this,
            displayName : "WebFabric",
            displayNamePrefix : "",
            displayNameSuffix : " web cluster",
            initialSize : 2,
            newEntity : webClusterFactory,
            httpPort : 8080,
            war: "/path/to/booking-mvc.war")
    public static void main(String[] argv) {
        TomcatFabricApp demo = new TomcatFabricApp(displayName : "tomcat example")
        BrooklynLauncher.manage(demo)
        JcloudsLocationFactory locFactory = new JcloudsLocationFactory([
                    provider : "aws-ec2",
                    identity : "xxxxxxxxxxxxxxxxxxxxxxxxxxx",
```

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

```
                        credential : "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",
                        sshPrivateKey : new File("/home/bob/.ssh/id_rsa.private"),
                        sshPublicKey : new File("/home/bob/.ssh/id_rsa.pub"),
                        securityGroups:["my-security-group"]
                    ])
            JcloudsLocation loc = locFactory.newLocation("us-west-1")
            JcloudsLocation loc2 = locFactory.newLocation("eu-west-1")
            demo.start([loc, loc2])
        }
    }
```

This creates a web-fabric. When started, this creates a web-cluster in each location supplied.

6.1.12 Examples Source

The source code for these examples is available for download from GitHub. To retrieve the source, execute the following command:

```
git clone git@github.com:cloudsoft/brooklyn-examples.git
```

You can also browse the code on the web: Brooklyn Examples on Github.

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

7 Understanding Management and Writing Policies

7.1 How Management Works

Brooklyn uses many of the ideas from autonomic computing to implement management of entities (including provisioning, healing, and optimising) in a structured and reusable fashion.

Each external system, process or service is represented as an entity within Brooklyn, with collections of these being represented and managed by other entities, and so forth through a hierarchy rooted in entities referred to as "Applications". Each entity has:

- provisioning and tear-down logic for the external system(s) it represents
- sensors which it publishes to report on its state and activity
- effectors which can be invoked to change it
- policies which perform analysis, enrichment (sensors), and execution (effectors). It is the policies in Brooklyn which perform the self-management (in autonomic terms) by monitoring sensors and invoking effectors.

The following recommendations should be considered when designing policies:

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)

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

ideally management should take run as a policy on the relevant entity

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 owner, and then cascade up. e.g. if the earlier VM CPU cannot be increased, the TOO_HOT event may go to the owner, 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

Best practices for this include:

- place management responsibility in policies at the entity, as much as possible
- place escalated management responsibility at the owner 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.
- order policies carefully, and mark sensors as "handled" (or potentially "swallow" them locally), so that subsequent policies and owner entities do not take superfluous (or contradictory) corrective action

For this milestone release, some of the mechanisms for implementing the above practices are still being developed.

7.1.1 Distributed Management

7.1.2 Resilience

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

7.1.3 Key APIs

- `ManagementContext` (Java management API)
- `EntityLocal` (used by policies)

7.1.4 Observing What is Happening

7.1.5.1 Management Web Console

Brooklyn comes with a web based management console that can be started using BrooklynLaucher:

```
public static void main(String\[\] argv) {
    application app = new MyApplicationExample(displayName:"myapp")
    brooklyn.launcher.BrooklynLauncher.manage(app)
    // ...
}
```

This will start an embedded Brooklyn management node, including the web console. The URL for the web console defaults to http://localhost:8081.

The mechanism for launching Brooklyn management will change in a future release. For this milestone release, the Brooklyn management node is embedded.

The Brooklyn Management Console serves as a way to track and manage Brooklyn entities. It contains two main views; *Dashboard* and *Details*.

The dashboard is a high level overview of the state of the application:

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

The details view gives an in depth view of the application and its entities. Child/parent relationships between the entities are navigable using the entity tree. Selecting a specific entity, allows you to access detailed information about that entity.

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

Brooklyn Webconsole – Detail

http://localhost:8081/detail/#summary

BROOKLYN Console | Dashboard | Detail

Tomcat Wide-Area Example Application > Fabric > AWS eu-west web cluster > DynamicWebAppCluster > TomcatServer

Summary | Sensors | Effectors | Activity | Location | Policies

name	description	value	last updated
host.address	Host IP address	79.125.88.127	2011-09-29 21:42:03
host.name	Host name	ec2-79-125-88-127.eu-west-1.compute.amazonaws.com	2011-09-29 21:42:03
http.port	HTTP port	8080	2011-09-29 21:42:03
jmx.context	JMX context path	jmxrmi	2011-09-29 21:42:03
jmx.port	JMX port	32199	2011-09-29 21:42:03
jmx.url	JMX URL	service:jmx:rmi:///jndi/rmi://ec2-79-125-88-127.eu-west-1.compute.amazonaws.com:32199/jmxrmi	2011-09-29 21:42:03
rmi.port	RMI port		2011-09-29 21:42:03
service.hasStarted	Service started	true	2011-09-29 21:43:28
service.isConfigured	Service configured	true	2011-09-29 21:42:03
service.state	Service lifecycle state	started	2011-09-29 21:42:03

Last update: 22:43:29

Summary: Description of the selected entity.

Sensors: Lists the attribute sensors that the entity has and their values.

Effectors: Lists the effectors that can be invoked on the selected entity.

Activity: Current and historic activity of the entity, currently running effectors, finished effectors.

Location: The geographical location of the selected entity.

Policies: Lists the policies associated with the current entity. Policies can be suspended, resumed and removed through the UI.

7.1.5.2 Security

In this milestone release only two Spring Security users are created: user and admin.

In future releases it will be possible to add an configure users.

admin access (username:admin, password:password).

user access (username:user, password:password).

Only the **admin** user has access to the Management Console.

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

7.1.6 Other Ways to Observe Activity

7.1.7 Java API

`ManagementContext` provides a Java programmatic API.

More information can be found in the javadoc for `ManagementContext`.

7.1.8.1 Command-line Console

Not available yet.

7.1.8.2 Management REST API

Not available yet.

7.1.8.3 Logging

This section is in development at the time of this milestone release.

Logging uses slf4j. Add the appropriate maven slf4j implementation dependency and logging config file.

Examples for testing can be found in some of the poms.

7.1.9 Sensors and Effectors

7.1.10.1 Sensors

Sensors are typically defined as static named fields on the Entity subclass. These define the channels of events and activity that interested parties can track remotely. For example:

```
/** a sensor for saying hi (illustrative), carrying a String value
    which is typically the name of the person to whom we are saying hi */
public static final Sensor<String> HELLO_SENSOR = ...
```

If the entity is local (e.g. to a policy) these can be looked up using get(Sensor). If it may be remote, you can subscribe to it through various APIs.

Sensors are used by operators and policies to monitor health and know when to invoke the effectors. The sensor data forms a nested map (i.e. JSON), which can be subscribed to through the `ManagementContext`.

Often `Policy` instances will subscribe to sensor events on their associated entity or its children; these events might be an `AttributeValueEvent` – an attribute value being reported on change or periodically – or something transient such as `LogMessage` or a custom `Event` such as "TOO_HOT".

Sensor values form a map-of-maps. An example of some simple sensor information is shown below in JSON:

```
{
  config : {
    url : "jdbc:mysql://ec2-50-17-19-65.compute-1.amazonaws.com:3306/mysql"
    status : "running"
  }
  workrate : {
    msgsPerSec : 432
  }
}
```

Sensor values are defined as statics which can be used to programmatically drive the subscription.

7.1.10.2 SubscriptionManager

This section is is progress at the time of this milestone release.

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

See the `SubscriptionManager` *class.*

7.1.10.3 Effectors

Like sensors and config info, effectors are also static fields on the Entity class. These describe actions available on the entity, similar to methods. Their implementation includes details of how to invoke them, typically this is done by calling a method on the entity. Effectors are typically defined as follows:

```
   /
** an effector which returns no value but which causes the entity to emit a HELLO sensor event */
   public static Effector<Void> SAY_HI = ...
```

Effectors are invoked by calling `invoke(SAY_HI, name:"Bob")` or similar. The method may take an entity if context is not clear, and it takes parameters as named parameters or a Map.

Invocation returns a `Task` object (extending `Future`) allowing the caller to understand progress and errors on the task, as well as `Task.get()` the return value (blocking).

The management framework ensures that execution occurs on the machine where the `Entity` is mastered, with progress, result, and/or any errors reported back to the caller. It does this through the `ExecutionManager` which, where necessary, creates proxy `Task` instances. The `ExecutionManager` associates `Tasks` with the corresponding `Entity` so that these can be tracked externally (and relocated if the Entity is remastered to a different location).

It is worth noting that, where a method corresponds to an effector, direct invocation of that method on an `Entity` will implicitly generate the `Task` object as though the effector had been invoked. For example, invoking `Cluster.resize(int)`, where `resize` provides an `Effector RESIZE`, will generate a Task which can be observed remotely.

The execution framework that provides this functionality is independent of Brooklyn, although it was developed for Brooklyn.

7.1.10.4 ExecutionManager

The `ExecutionManager` is responsible for tracking simultaneous executing tasks and associating these with given **tags**. Arbitrary tasks can be run by calling `Task submit(Runnable)` (similarly to the standard `Executor`, although it also supports `Callable` arguments including Groovy closures, and can even be passed `Task` instances which have not been started). `submit` also accepts a few other named parameters, including `description`, which allow additional metadata to be kept on the `Task`. The main benefit then is to have rich metadata for executing tasks, which can be inspected through methods on the `Task` interface.

By using the `tag` or `tags` named parameters on `submit` (or setting `tags` in a `Task` that is submitted), execution can be associated with various categories. This allows easy viewing can be examined by calling `ExecutionManager.getTasksWithTag(...)`.

In this example uses Groovy, with time delays abused for readability. Brooklyn's test cases check this using mutexes, which is recommended. :

```
   ExecutionManager em = []
      em.submit(tag:"a", description:"1-a", { Thread.sleep(1000) })
      em.submit(tags:["a","b"], description:"2-a+b", { Thread.sleep(1000) })
      assert em.getTasksWithTag("a").size()==2
      assert em.getTasksWithTag("a").every { Task t -> !t.isDone() }
      Thread.sleep(1500)
      assert em.getTasksWithTag("a").size()==2
      assert em.getTasksWithTag("a").every { Task t -> t.isDone() }
```

Note that it is currently necessary to prune dead tasks, either periodically or by the caller. By default they are kept around for reference. It is expected that an enhancement in a future release will allow pruning completed/ failed tasks after a specified amount of time.

It is possible to define ParallelTasks and SequentialTasks and to specify inter-task relationships with TaskPreprocessors — e.g. either submitting a SequentialTasks or specifying

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

`em.setTaskPreprocessorForTag("a", SingleThreadedExecution.class)` will cause `2-a+b` to run after `1-a` completes. This allows building quite sophisticated workflows relatively easily. For more information consult the javadoc on these classes and associated tests.

7.1.11 Writing Policies

This section is not complete in this milestone release.

- Policies often run periodically or on sensor events
- Policies can add subscriptions to sensors on any entity (although usually it will be its related entity, those entities it owns, and/or those entities which are members)
- Policies may invoke effectors (management policies) or simply generate new attributes or events (enricher policies).

7.1.12.1 Implementation Classes

This section is not complete in this milestone release.

- extend `AbstractPolicy`, or override an existing policy

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

8 Writing Custom Entities

8.1 Custom Entity Development

What you need to know to create new custom app components or groups as Brooklyn entities

8.1.1 The Entity Lifecycle

- importance of serialization, ref to How Mgmt Works
- Ownership (children) and Membership (groups)

8.1.2 What to Extend -- Implementation Classes

- entity implementation class hierarchy
    - `SoftwareProcessEntity` as the main starting point for base entities (corresponding to software processes)
    - cluster, group, stack, fabric, etc. provide conveniences for collecting entities (including software processes)
- traits (mixins, otherwise known as interfaces with statics) to define available config keys, sensors, and effectors; and conveniences e.g. StartableMethods 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)

8.1.3 Configuration

- TODO: why to use config?
- AttributeSensorAndConfigKey fields can be automatically converted, for `SoftwareProcessEntity` this is done in `preStart()` (for other entities it must be done manually if required)
- 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 (in brief, it allows e.g. "8080-8099,8800+" to at 8080, try sequentially through 8099, then try from 8800 and try until all ports are exhausted); this is particularly useful on a contended machine (such as localhost!), and of course the config is done by the user like ordinary configuration, and the actual port used is reported back as a sensor on the entity

8.1.4 Implementing Sensors

- e.g. HTTP, JMX

Sensors at base entities are often retrieved by adapters which poll the entity's corresponding instance in the real world. The `SoftwareProcessEntity` 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()
```

def http = sensorRegistry.register(new HttpSensorAdapter(mgmtUrl, period: 200*TimeUnit.MILLISECONDS)) http.poll(SERVICE_UP, { responseCode==200 }) http.suburl("requests").poll(REQUEST_COUNT) http.suburl("requestDurationsAsJsonList").poll(MAX_PER_SITE) { (json.durations as List).collect({ it as Long }).max() } }

(where we've imagined `url+"/requests` serves up the request count as a string, and `url+"/requestDurationsAsJsonList" returns a JSON string, which the adapter's` json `field in the`

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

closure lets us access as a map; responseCode is the HTTP status response code for the request, and other fields in HttpResponseContext`` are also available, including headers, content, and errors)

Note the first line; 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.

8.1.5 Implementing Effectors

The `Entity` implementation defines the sensors and effectors available, the wiring for the sensors, and in simple cases it may be straightforward to capture the behaviour of the effectors in methods. For example deploying a WAR to a cluster can be done as follows:

```
TODO
```

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 For example, 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, such as passing JMX environment variables to Java over SSH, can then be inherited in the driver class hierarchy, for example JavaStartStopSshDriver extends StartStopSshDriver and parents JBoss7SshDriver.

8.1.6 Implementing Policies

This section is in development at the time of this milestone release. Please see the class *`brooklyn.policy.Policy`* and implementations.

8.1.7 Testing

- Run in a mock `SimulatedLocation`, defining new metaclass methods to be able to start there and assert the correct behaviour when that is invoked

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

9 Brooklyn Extras

9.1 Systems Available Out-of-the-Box

Brooklyn comes bundled with support for a large number of systems and entities.

Please note that in this pre-release version not all are fully functional. The documentation in this section is also in progress. Please contact Cloudsoft if further assistance is required.

9.1.1 Web

9.1.2.1 JavaWebAppServer

Currently Tomcat and JBoss are supported. For instantiating an instance of Tomcat see TomcatServer. For JBoss, depending on version refer to JBoss7Server or JBoss6Server.

9.1.2.2 Nginx

Nginx provides clustering support for several web/app servers.

The install process downloads the sources for both the service and the sticky session module, configures them using GNI autoconf and compiles them. This requires gcc and autoconf to be installed. The install script also uses the yum package manager (if available) to install openssl-devel which is required to build the service. This will only work on RHEL or CentOS Linux systems, but the install process should proceed on a vanilla system with development tools available.

On debian/ubuntu to build nginx you can get the required libraries with: apt-get install zlib1g-dev libdigest-sha-perl libssl-dev

9.1.3 Database

9.1.4.1 Apache Derby

Support for Apache Derby which is a pure-Java SQL database. For setting up an instance of a server see DerbySetup.

9.1.5 NoSQL

9.1.6.1 Redis

Redis is a distributed key-value store. We support master/slave replication of a store as a clustered cache. This gives a series of read-only slaves and a single read-write master, which propagates to the slaves with eventual consistency. See RedisSetup.groovy.

9.1.6.2 Infinispan

Support for Infinispan server is currently in progress.

9.1.6.3 Gemfire

Gemfire support is a current work in progress providing capability to configure and setup Gemfire servers and define clusters. See GemfireSetup and GemfireCluster.

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

9.1.7 Messaging

9.1.8.1 Qpid

Qpid support provides a JMS broker, running over AMQP. This exposes JMS queues and topics as entities as well. See QpidSetup for instantiating a broker.

9.1.8.2 ActiveMQ

ActiveMQ support provides a JMS broker. This exposes JMS queues and topics as entities as well. See ActiveMQSetup for instantiating a broker.

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

10 Cloudsoft Developer License

10.1 CLOUDSOFT CORPORATION LTD END USER LICENSE AGREEMENT

BEFORE USING THE CLOUDSOFT CORPORATION ("CLOUDSOFT") SOFTWARE ("SOFTWARE"), ANY ASSOCIATED DOCUMENTATION ("DOCUMENTATION", TOGETHER WITH THE SOFTWARE, THE "PRODUCT") OR ANY OTHER MATERIALS OR THIRD PARTY SOFTWARE PROVIDED WITH THE SOFTWARE, READ CAREFULLY THE FOLLOWING TERMS AND CONDITIONS OF THIS END USER LICENSE AGREEMENT ("AGREEMENT") BETWEEN CLOUDSOFT AND THE INDIVIDUAL PERSON OR ENTITY ("LICENSEE") RECEIVING THE SOFTWARE TO WHICH THIS AGREEMENT IS ATTACHED, OR INTO WHICH THIS AGREEMENT IS EMBEDDED. BY USING THE PRODUCT OR ANY PART THEREOF, OR BY CLICKING "I ACCEPT" BELOW, YOU ACKNOWLEDGE THAT YOU HAVE READ, UNDERSTOOD AND AGREE TO BE BOUND BY THE TERMS OF THIS AGREEMENT AND THAT YOU HAVE THE ABILITY TO BIND THE ENTITY RECEIVING THE PRODUCT. IF YOU DO NOT AGREE WITH THE TERMS AND CONDITIONS OF THIS AGREEMENT, YOU MAY NOT INSTALL OR USE THE PRODUCT. NOTWITHSTANDING THE FOREGOING, IF LICENSEE HAS ENTERED INTO ANOTHER VALID AGREEMENT WITH CLOUDSOFT THAT COVERS THE PRODUCT (WHETHER OR NOT SUCH VALID AGREEMENT WAS PHYSICALLY EXECUTED BY LICENSEE, INCLUDING, WITHOUT LIMITATION ANY "CLICK-THROUGH" AGREEMENT OR ANY APPLICABLE TERMS AND CONDITIONS POSTED ON ANY CLOUDSOFT WEBSITE), THEN THE FOLLOWING LICENSE AGREEMENT TERMS ARE SUPERSEDED BY SUCH OTHER VALID AGREEMENT, AND DO NOT APPLY TO LICENSEE'S USE OF THE PRODUCT. IN ADDITION, THIS LICENSE AGREEMENT APPLIES ONLY TO THE PRODUCT AND DOES NOT APPLY TO ANY THIRD PARTY SOFTWARE, INCLUDING OPEN SOURCE COMPONENTS, AS MAY BE LISTED IN THE LICENSES DIRECTORY OR SPECIFIED IN THE DOCUMENTATION. SUCH THIRD PARTY SOFTWARE IS LICENSED UNDER THE TERMS OF THE APPLICABLE THIRD PARTY LICENSE AGREEMENT LISTED IN THE LICENSES DIRECTORY OR SPECIFIED IN THE DOCUMENTATION.

1. License Grant. Subject to the terms and conditions of this Agreement, Cloudsoft agrees to grant, and does hereby grant to Licensee during the term of this Agreement, a limited, non-exclusive, non-transferable right and license, solely to the object code version of the Software, and to the Documentation and any other materials provided to Licensee by Cloudsoft hereunder ("Materials"), without the right to grant or authorize sublicenses or to further distribute the Product or Materials, to install the Software on computers owned or leased by Licensee and to use the Product and Materials solely for Licensee's internal business operations, development, evaluation, and educational purposes ("Permitted Uses"). The Product and Materials may not be used for any purpose other than for Permitted Uses, and may not be used by any other person or entity other than Licensee. Licensee may make up to two copies of the Product for backup and/or archival purposes.
2. License Restrictions. Licensee agrees not to: (a) copy or use the Product in any manner except as expressly permitted in this Agreement; (b) transfer, sell, rent, lease, distribute, or sublicense the Product to any third party; (c) use the Product for providing time-sharing services, service bureau services or as part of an application services provider or as a service offering; (d) reverse engineer (except as permitted by applicable law), disassemble, decompile the Products; (e) alter modify, enhance or prepare any derivative work from, the Product; (f) alter or remove any proprietary notices in the Product; or (g) make available to any third party any analysis of the results of operation of the Product, including benchmarking results, without the prior written consent of Cloudsoft.
3. Ownership. The Product and Materials are and shall remain the sole property of Cloudsoft and its licensors, and, except as expressly provided herein, Cloudsoft and its licensors retain all right, title and interest in and to the Product or Materials, including all intellectual property rights therein and thereto.
4. Requirements of Licensee. Licensee may provide feedback regarding the Product including without limitation any functionality issues, and errors, flaws, failures, or faults in the Product (collectively, "Feedback") via an online Cloudsoft forum or other method. Licensee hereby grants to Cloudsoft a

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

perpetual, irrevocable, worldwide, sublicensable, transferable, royalty-free, fully-paid, right and license to use and exploit in any manner and for any purpose all Feedback and related information.

5. Confidential Information. The Product and Materials contain Confidential Information and trade secrets of Cloudsoft and its licensors. "Confidential Information" means all software code and information furnished by Cloudsoft in oral, written or machine-readable form, disclosed as a result of this Agreement, and that should reasonably have been understood by Licensee, because of legends or other markings, the circumstances of disclosure or the nature of the information itself, to be proprietary and confidential to Cloudsoft, a Cloudsoft affiliate or other third party. Licensee will use the same standard of care to prevent unauthorized access to or disclosure of the Confidential Information that Licensee uses to prevent the disclosure of its own similar confidential information, but in no event less than a reasonable standard of care. Licensee will disclose the Confidential Information only to its employees with a need to know for the purposes of this Agreement. The restrictions of this Agreement on use and disclosure of Confidential Information shall not apply to information that becomes publicly known through no fault of the Licensee or its personnel. All obligations regarding Confidential Information received prior to the expiration or termination of this Agreement shall survive the expiration or termination of this Agreement.
6. Warranty Disclaimer. THE PRODUCT AND THE MATERIALS ARE PROVIDED "AS IS" AND WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY, TITLE, NON-INFRINGEMENT, QUIET ENJOYMENT, ACCURACY OF DATA, SYSTEM INTEGRATION, COURSE OF PERFORMANCE AND FITNESS FOR A PARTICULAR PURPOSE. CLOUDSOFT DOES NOT GUARANTEE OR WARRANT THAT THE USE OF THE PRODUCT WILL BE UNINTERRUPTED OR ERROR FREE.
7. Limitation of Liability. IN NO EVENT WILL CLOUDSOFT BE LIABLE FOR ANY CLAIM BASED UPON A THIRD PARTY CLAIM, OR ANY INCIDENTAL, CONSEQUENTIAL, SPECIAL, INDIRECT, EXEMPLARY OR PUNITIVE DAMAGES, WHETHER ARISING IN TORT, CONTRACT, OR OTHERWISE; OR FOR ANY DAMAGES ARISING OUT OF OR IN CONNECTION WITH ANY MALFUNCTIONS, DELAYS, LOSS OF DATA, LOST PROFITS, LOST SAVINGS, INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS, EVEN IF CLOUDSOFT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. CLOUDSOFT'S AGGREGATE AND CUMULATIVE LIABILITY ARISING OUT OF OR RELATING TO THIS AGREEMENT, REGARDLESS OF THE FORM OF THE CAUSE OF ACTION, WHETHER IN CONTRACT, TORT (INCLUDING WITHOUT LIMITATION NEGLIGENCE), STATUTE OR OTHERWISE WILL BE LIMITED TO DIRECT DAMAGES AND WILL NOT EXCEED ONE THOUSAND DOLLARS (US $1,000). THE ALLOCATIONS OF LIABILITY IN THIS SECTION 7 REPRESENT THE AGREED AND BARGAINED FOR UNDERSTANDING OF THE PARTIES, AND THE COMPENSATION OF CLOUDSOFT FOR THE SERVICES PROVIDED HEREUNDER REFLECTS SUCH ALLOCATIONS. THE FOREGOING LIMITATIONS, EXCLUSIONS AND DISCLAIMERS ARE AN ALLOCATION OF THE RISK BETWEEN THE PARTIES AND WILL APPLY TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, EVEN IF ANY REMEDY FAILS IN ITS ESSENTIAL PURPOSE.
8. Term and Termination. The term of this Agreement will begin on the earlier of the date this Agreement is accepted by Licensee or Licensee's first use of the Product and end on the date this Agreement is terminated by either party. This Agreement may be terminated at any time by either party upon written notice to the other party. Upon termination or expiration of this Agreement, Licensee will use reasonable efforts to deinstall and destroy the Product or return the Product and Materials to Cloudsoft. Termination will not affect any claim, liability or right arising prior to termination. All rights and obligations granted under Sections 2, 3, 5, 6, 7, 8, 9 and 10 of this Agreement will survive the expiration or termination of this Agreement.
9. Government Rights. The Products under this Agreement are "commercial computer Products" as that term is described in DFAR 252.227-7014(a)(1). If acquired by or on behalf of a civilian agency, the U.S. Government acquires this commercial computer Products and/or commercial computer Products documentation subject to the terms and this Agreement as specified in 48C.F.R. 12.212 (Computer Products) and 12.11 (Technical Data) of the Federal Acquisition Regulations ("FAR") and its successors. If

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.

acquired by or on behalf of any agency within the Department of Defense ("DOD"), the U.S. Government acquires this commercial computer Products and/or commercial computer Products documentation subject to the terms of this Agreement as specified in 48 C.F.R. 227.7202 of the DOD FAR Supplement and its successors. Licensee will not export the Products in violation of the export laws of the United States or of any other country.

10 General. This Agreement constitutes the entire agreement between the parties concerning the subject matter hereof, notwithstanding any different or additional terms that may be contained in the form of purchase order or other document used by Licensee to place orders or otherwise effect transactions hereunder, which such terms are hereby rejected. This Agreement supersedes all prior or contemporaneous discussions, proposals and agreements between the parties relating to the subject matter hereof, provided that if Licensee has entered into or later enters into another valid agreement with Cloudsoft regarding the Product (whether or not such other valid agreement was physically executed by Licensee, including without limitation any "click-through" agreement or any applicable terms and conditions posted on any Cloudsoft website), then such other agreement shall supersede this Agreement. No amendment, modification or waiver of any provision of this Agreement will be effective unless in writing and signed by both parties. If any provision of this Agreement is held to be invalid or unenforceable, the remaining portions will remain in full force and effect and such provision will be enforced to the maximum extent possible so as to effect the intent of the parties and will be reformed to the extent necessary to make such provision valid and enforceable. No waiver of rights by either party may be implied from any actions or failures to enforce rights under this Agreement. Neither party will be liable to the other for any delay or failure to perform due to causes beyond its reasonable control (excluding payment of monies due). Unless otherwise specifically stated, the terms of this Agreement are intended to be and are solely for the benefit of Cloudsoft and Licensee and do not create any right in favor of any third party. This Agreement will be governed by and construed in accordance with the laws of the United States and the State of California without reference to its conflict of laws principles. All disputes arising out of or relating to this Agreement will be submitted to the non-exclusive jurisdiction of a court of competent jurisdiction located in the Northern District of California or a state court located in San Francisco, California, and each party irrevocably consents to such personal jurisdiction and waives all objections to this venue. All notices must be in writing and will be effective three (3) days after the date sent.

©2012, Cloudsoft Corporation • ALL RIGHTS RESERVED.