Cattle Documentation

Release 0.1

Darren Shepherd

September 16, 2014

Contents

Contents:

CHAPTER 1

Installation

Contents:

1.1 Quick Installation

Follow the instructions at instructions at https://github.com/cattleio/cattle/blob/master/README.md

1.2 Detailed Installation

These instructions are a bit more granular than *Quick Installation* but allow you to install and run Cattle in contexts where Docker does not exist.

For a simple installation you have two basic things that need to be setup, the management server and a hypervisor/Docker server. They can be the same physical/virtual machine if you want.

1.2.1 1. Download

If you are installing using Docker, you do not need to download anything. If you are just running Java manually, then you must download the `cattle.jar` from the Releases Page

1.2.2 2. Management Server Installation

You can either run Cattle within docker or manually. It is really a matter of preference which approach you take. Running in docker has the advantage of not having to install Java and pretending that it doesn't exist.

2a. The "Docker Way"

```
docker run -p 8080:8080 cattle/server
```

NOTE: This will use port 8080 on the host. If port 8080 is not free, you can choose a different port by doing '-p 8081:8080' to use port 8081, for example. Or if you want to just allocate a random port '-p 8080'.

2b. Manually

Java 6+ is required, Java 7+ is recommended.

```
java -jar cattle.jar
```

1.2.3 3. Make sure it's running

It may take up to 30 seconds to startup the first time. Once Cattle is running you should see the below message

> Startup Succeeded, Listening on port 8080

If you see tons of Java stack traces, then something bad happened. If you can hit http://localhost:8080 and it serves up the UI you should be good to go.

NOTE: If you are running the management server under Docker the listen port might not be 8080 but instead the random port that was chosen by Docker. Run "docker ps" to see which port is mapped to 8080.

1.2.4 4. Registering a Hypervisor/Docker Server

Docker v0.9.1+ is required to be installed.

From the hypervisor or Docker server run the following commands to register the node with management server replacing HOST:PORT with the host and port of the mangement server. This is the same port that you used to access the UI.

```
curl http://<HOST:PORT>/v1/authorized_keys | sudo tee -a /root/.ssh/authorized_keys
curl -X POST http://<HOST:PORT>:8080/v1/agents
```

When Cattle first starts it generates a SSH key. The above commands will download the public key and then register itself with the Cattle management server. The default communication transport between the management server and the host is SSH. There are other approaches, but SSH is the simplest and most secure approach for Linux based hosts.

1.2.5 5. Confirm registration worked

If you hit http://localhost:8080/v1/agents you should see one agent and the state should be "active." If the state is not active within five minutes, then it probably didn't work.

If the agent is active go to http://localhost:8080/v1/hosts and within one minute you should see one host that is active.

1.2.6 6. Run something

You can now open the UI to http://localhost:8080/v1/containers and click "Create." Enter a imageUuid in the format "docker:name" for example "docker:cattle/server" or "docker:ubuntu" and a command, like "sleep 120."

1.3 boot2docker Installation

1.3.1 Install boot2docker

Refer to https://github.com/boot2docker/boot2docker for instructions to install boot2docker.

The super short instructions for installing boot2docker on OS X are:

```
brew update
brew install boot2docker
boot2docker init
boot2docker up
```

1.3.2 Log into boot2docker

Log into boot2docker with **boot2docker ssh -L 8080:localhost:8080**. This will forward port 8080 from your laptop/desktop to the VirtualBox VM. boot2docker does not handle setting up port forwards, so adding **-L 8080:localhost:8080** is essential if you wish to access the API/UI of Cattle.

```
boot2docker ssh -L 8080:localhost:8080
```

The password is tcuser

1.3.3 Start Cattle

```
docker run -d -p 8080:8080 cattle/server
```

1.3.4 Setup SSH Keys

```
mkdir -p /home/docker/.ssh
chmod 700 /home/docker/.ssh
curl -s http://localhost:8080/v1/authorized_keys | tee -a /home/docker/.ssh/authorized_keys
```

1.3.5 Register Server

```
curl -X POST http://localhost:8080/v1/agents -F user=docker
```

It may take a couple minutes to register the server because the server must pull a special image to run the agent under.

1.3.6 Check Status

If something doesn't seem to be working you can look at the logs with **docker logs -f CONTAINER_ID**. If everything worked correctly you should see a host named boot2docker at http://localhost:8080/static/.

1.4 CoreOS Installation

Installation on CoreOS is pretty straight forward. You will basically will follow the *Quick Installation*.

1.4.1 Run in Docker

The key things to note is that you must run Cattle in Docker, you're not going to install Java is CoreOS. Specifically, you are going to be starting the Cattle management server as follows

```
docker run -d -p 8080:8080 cattle/server
```

1.4.2 Adding a Server

When registering a server you need to specify that the user will be the core user and not the root user.

Don't put the SSH key in /home/core/.ssh/authorized_keys because that file may later be clobbered by update-ssh-keys used in CoreOS. Instead use the update-ssh-keys script to register the key. Additionally, when adding the agent you need specify the core user also. Run the below commands instead of the commands in the *Quick Installation*.

```
curl -s http://<HOST:PORT>/v1/authorized_keys | update-ssh-keys -A cattle
curl -X POST http://<HOST:PORT>/v1/agents -F user=core
```

1.5 AWS CloudFormation Installation

To get a sample installation up and running real fast you can use CloudFormation, just follow one of the links below.

1.5.1 Regions

- us-east-1
- us-west-1
- us-west-2
- eu-west-1
- sa-east-1
- ap-southeast-1
- ap-southeast-2
- ap-northeast-1

Warning: Make sure you know your SSH keyname for the appropriate region and enter it correctly.

1.5.2 Information

This CloudFormation template will create a single EC2 instance that has Cattle installed and it will register itself as a docker and libvirt hypervisor. Once the installation is done, which could take 20 minutes, the UI and API will be accessible at http://ec2-XX-XXX-XXX-XX.us-west-1.compute.amazonaws.com:8080/v1.

1.5.3 Logging in

You can log into the server by running ssh similar to below

```
ssh -l ubuntu -i mykey.pem ec2-XX-XXX-XXX-XX.us-west-1.compute.amazonaws.com
```

The installation log is at `/var/log/cattle-install.log`. If everything ran successfully you should be able to run `cattle list-host` and see two hosts registered, one for Docker and one for Libvirt.

CHAPTER 2

Design

2.1 Vision

Using the analogy of a Cloud Operation System, Cattle would be the kernel. Xen, KVM, Docker, Ceph would be the drivers. EC2, GCE, OpenStack, and CloudStack APIs would be glibc. User space would be the plethora of tools and applications that consume Cloud APIs.

Cattle is first and foremost a lightweight, flexible, and reliable orchestration engine. The umbrella of Cattle is purposely intended to not reach too far. The goal of the platform is to stay focuses on building the core building blocks needed to assemble a full scale IaaS solution. As one moves up the stack in functionality, by design, it should be less and less specific to Cattle.

As one writes a simple server side application today, whether they are running on Linux, Solaris, or FreeBSD is largely inconsequential. As long as your runtime, such as Python, Node.js, or Ruby, is supported, the details of the underlying technology are often abstracted away. As such is the vision of Cattle. While it is design to run the largest and most complex clouds that can exist, it is intended that most application will be abstracted away from Cattle by either an API such as EC2 or a platform such as CloudFoundry.

Cattle aims to fit into a larger ecosystem of cloud and infrastructure tools that already exist today. The intention is not to build a new ecosystem around Cattle, but instead be a member of a larger ecosystem by leveraging the standards (official or de-facto) that already exist. With this in mind, being architecturally compatible with APIs such as EC2 is a top priority.

2.2 Principles

2.2.1 Simplicity

Orchestrating infrastructure can be a relatively complex task. There are a lot of moving parts and failures in systems are abundant and expected. Cattle aims to make this complex problem simple. To be frank, this is easier said than done. It is not enough in an IaaS system to say that simplicity is at the level of the user interface (whether that be UI or API). Simplicity must extend further down into the core of the system. The fundamental truth of IaaS is that no two clouds are the same. This is not a flaw or shorting coming of IaaS and should not be viewed as a “snowflake” problem.

Take the networking world as an example. A vendor does not go to a customer and say every network should be exactly the same. Instead they sell switches, routers, and other devices to assemble and create their own networks. The reality though is that best practices and reference architectures arise. But even when one implements a reference architecture, it is still based on a reference and variations will occur. The compute and storage world have a similar story with reference architectures and variations. IaaS aggregates networking, storage and compute into a single solution and thus the variations become multiplicative. Overtime, reference IaaS architectures and best practices will emerge.

Cattle focuses on simple patterns in architecture and design in order to deal with the complex variations that will occur. Consistently throughout the architecture of Cattle simple proven patterns are chosen over more complex patterns that are theoretically more complete. The theorical solutions are left as a future optimization that may or may not be ever needed. While simple and possibly non-comprehensive patterns are chosen it is not done naively. All patterns are chosen knowing their limitations and what steps could be taken if those limitations become an issue.

2.2.2 Scalability

The scalability goals of Cattle are similar to most other IaaS stacks. Cattle should scale to millions of devices and 100s of millions user resources (virtual machines) within a single control plane. Billions and beyond will be possible with multiple federated control planes. The difference with Cattle is that while that extreme scalability is important, we do not ignore the fact that the vast majority of clouds are less than 100 servers.

Aligned with the principle of simplicity, Cattle should be the easiest stack to run and should require a minimal amount of resources to run a cloud of a hundred servers. A cloud of that size can be controlled by a laptop or a simple ARM device.

In order to meet high scalability requirements, particular attention is given to ensure that as devices, such as hypervisors, are added, that the resources they require are horizontally scalable. For example, if a hypervisor holds a connection to a database, then that will not scale to millions because the database can not have that many connections.

2.2.3 Reliability

Orchestration should never fail.

The nature of infrastructure orchestration is that systems fail. From a computer science perspective, systems fail quite often. Additionally, their is a growing trend to build infrastructure on commodity hardware; commodity hardware often being synonymous with less reliable.

While the systems being orchestrated will fail, the orchestration itself should never fail. This does not mean that all user requests will succeed. If a server fails and their is not enough compute available, the request should result in an error. The IaaS system will detect the insufficient capacity and systems failure and the user request will result in an error. The orchestration succeeded, as the system did what it should do, but the operation resulted in a error.

To further expound on this point, orchestration should continue in the event of any system failure, whether that failed system is being orchestrated or it is part of the orchestration system itself. This means if the orchestration management server dies while it is in the middle of doing an operation, that operation should continue if there is another management server available, or if not, when that server comes back online.

At the heart of Cattle, all architecture decisions are made with consideration to how one can reliability orchestrate knowing that all underlying systems are unreliable.

CHAPTER 3

Architecture

Contents:

3.1 Logical System Architecture

Overview

Cattle system architecture is separated into the following logical servers.

3.1.1 Database

Cattle is based on a RDBMS. The currently support databases are MySQL, H2 (Java), and HSQLDB (Java). Other databases such as PostgreSQL and SQL Server will be supported in the future. Cattle is built upon jOOQ and Liquibase to abstract the database. Any database that is supported by both of these frameworks should eventually be supported by Cattle. Since the abstraction provided by these libraries is not perfect, the limiting factor in adopting other databases is the required testing.

3.1.2 Lock Manager

A distributed lock manager (DLM) is used to control concurrent access to resources. Cattle currently supported Hazelcast and Zookeeper as a lock manager.

3.1.3 Event Bus

Communication between distributed components is done using an event bus. The messaging style used in Cattle can be best compared to UDP multicast. There is no assumptions of reliability or persistence in the messaging transport. This means that from an operational standpoint the messaging system is not considered a repository of state. If you were to do maintenance one could completely shutdown, delete, and reinstall the messaging servers with no impact except that operations will pause until the messaging system is back online. The currently supported messaging systems are Redis and Hazelcast.

3.1.4 API Server

The API server accepts the HTTP API requests from the user. This server is based on a Java Servlet 3.0 container. By default Cattle ships with Jetty 8 embedded, but it is also possible to deploy Cattle as a standard war on any Servlet Container such as Tomcat, WebLogic, or WebSphere.

Note: The current dashboard UI uses WebSockets and the current Cattle WebSocket implementation is Jetty specific. This is concidered a flaw and should be fixed to work on any servlet container, or the dashboard should support line terminated event polling.

The API server only needs the database and lock manager to be available to service requests. This means for maintanence all other back-end services can be completely stopped if necessary. The API server is stateless and does not do any orchestration logic. All orchestration logic is deferred to the back-end servers. Once a user receives a 201 or 202 HTTP response, the remaining logic will be preformed elsewhere.

3.1.5 Process server

The process server can be viewed as the main loop of the system. It is responsible for controlling the execution of processes. If the process server is stopped no requests will be executed until it is brought back online.

3.1.6 Agent Server

The agent server is responsible for the communication to and from the remote agents. Remote agents have no access to the database, the event bus, or lock manager. The agent server presents a HTTP REST API that the remote agents can use to publish and subscribe to the event bus. This proxy serves two purposes. First, it is used as a security mechanism to control which events are accessible to the remote agent. Second, it is used as a means of scaling the cloud. When scaling number of hosts, much of the scaling concerns have to do with persistent socket connections. By putting an intermediate server between the agent and the event bus, this makes scaling the hosts move of a factor of O(n^2) and not O(n).

3.1.7 Remote Agents

Remote agents can be developed in any language. The default agent is developed in python and intended to run on a Linux host. Agents are designed to be dumb. They do not initiate an action by themselves and have no ability to call out and get information. When a command is sent to an agent all of the data that is needed for the action is encapsulated in the command. The agent then performs the request and returns the result. All operations are done in a request, reply fashion, with all requests initiated by the core orchestration system and never the agent.

3.2 Deployment Architecture

Cattle can be deployed in many configurations to match the scale and needs of the deployed cloud. The smallest configuration is a single process and the largest is a fully distributed set process with external system services such as MySQL, Redis, and Apache ZooKeeper.

3.2.1 Server Profiles

Cattle is distributed as a single jar file. All the code needed to run Cattle is in that single file. When the JVM is launched you can specify a server profile that with determine if the JVM will run as all servers or specific one. For example, if you run with `-Dserver.profile=api-server` the JVM will only run as an API server. Additionally, based on configuration you can tell Cattle which system services to use such as Hazelcast or Apache ZooKeeper for locking.

3.2.2 Reconfiguring Environment

If you start with the simplest deployment scenario (a single JVM) you can alway reconfigure the environment to more complex deployment architectures without requiring a reinstall. This means you can start very simple and then you can adaptive to the needs of your cloud as things change.

3.2.3 Example Deployment Scenarios

Single JVM

Great for just starting out or if you just want a simple step. This setup will scale to meat the vast majority of dev/test clouds that exist today.

Redundant JVM

In this scenario you'd like a bit of redundancy such that if a single process is down, the entire management stack is not down. For this scenario we switch to an external database and then use Hazelcast to do locking and eventing.

Distributed services

In this scenario you'd like to break out the JVM components into distributed components such that you can scale and maintain them independently. The API server, process server, and agent server all have different scaling characteristics.

The process server is used to throttle the concurrency and throughput of the entire system. By adding more process serves you effectively 2x, 3x, 4x, etc the concurrency of the system. More is not necessarily better. The down stream systems such as the hypervisors and storage devices still run at a constant speed. If you increase the concurrency such that downstream systems can not handle the work, you will be effectively wasting a lot of resources, because the majority of the process jobs will have to be rescheduled because the downstream system is busy.

The API server is scaled based on the amount of traffic from users. If you have a significant amount of reads, for example, you may want to spin up some more API servers and point them to read slaves. Or, imagine you have a UI. There is a good reason to run API servers dedicated to the UI and then separate API servers dedicated to the public API. If your public API gets hammered the UI will still be responsive (assuming that the public api is being properly throttled and not overwhelming the DB).

The number of agent servers are scaled in a ratio to match the number of hypervisors. A persistent connection is maintained between the agent server and the agents to allow efficient management of the hypervisors. As a result, you probably only want about 10,000 hypervisors per agent server. An agent server can manage the connection of any agent in its assigned agent group. Agent groups are used as a means of grouping and controlling the amount of connections managed by an agent server.

Concepts

Contents:

4.1 Orchestration

This section describes the underlying concepts on how orchestration is carried out in Cattle.

4.1.1 Resources

The base unit in Cattle is a resource. Resources are opaque objects that are orchestrated. What a resources is and what it represents is largely unknown to the core Cattle orchestration system. The core orchestration system mostly cares about the state, type, and relationship of the resources.

An example of a resource is a volume or IP address. Instance is the most important resource as it is the generalized concept of either a container or virtual machine.

You can list all the defined resources at http://localhost:8080/v1/resourcedefinitions in the API.

4.1.2 Processes

Once you have defined resources, you can attach processes to the resources. Processes are what drive the orchestration logic. You can list all the defined processes at http://localhost:8080/v1/processdefinitions in the API. One important process is the "instance.start" process which is responsible for starting a virtual machine or container. For any process you can navigate to the process definition in the API and then follow the "resourceDot" link to get a visual respentation of the process. For example the "instance.start" process is avaiable at http://localhost:8080/v1/processdefinitions/1pd!instance.start/processdot and looks like below.

Process are defined as having a start, transitioning, and done state. All processes are modeled in this fashion. For a process to be started, it obviously must be in the start state. Once the process has been started it will update the state to the transitioning state. For a process to be completed it must move from the transitioning state to the done state. If there is not logic attached to that transition, the orchestration system will simply update the state and be done.

To make the orchestration system actually perform real operations, you must attach some logic. In the above diagram you can see that the "InstanceStart" logic has been attached to this process. One can attached any logic they choose.

This makes the orchestartion system very flexible and extensible. At any point in the lifecycle of the resources you can plug-in any arbitrary logic.

Once you assign enough states and process to a resource you begin to construct a finite state machine for the resource. You can combine all the processes for a resource into a single view by navigating to the resourceDot link of a resourceDefinition. For example, the combined process diagram for an instance is available at http://localhost:8080/v1/resourcedefinitions/1rd!instance/resourcedot and looks similar to below.

4.1.3 Process Handler

Process handlers are the way in which you attach logic to a process. At the lowest level they are Java code that implement the `io.cattle.platform.engine.handler.ProcessHandler` interface. Instead of writing Java code, you can instead have the orchestration system send out an event that you process handler will respond too. This means you can add logic in any programming or scripting language. Refer to *Simple External Process Handler* for an example of registering an external handlers and how to reply to events.

Configuration

Contents:

5.1 Configuring Cattle

The configuration of Cattle is very flexible. Under the hood it is based on Netflix's Archaius which is in turn based on Apache Commons Configuration. The configuration of Cattle is essentially key value based. The key values pairs will be read from the following sources. If the configuration is found in the source it will not continue to look for it in further sources. The order is as below:

1. Environment variables
2. Java system properties
3. `cattle-local.properties` on the classpath
4. `cattle.properties` on the classpath
5. The database from cattle.setting table

The current configuration can be viewed from and modified from the API at http://localhost:8080/v1/settings. Any changes done using the API will saved in the database and immediately refreshed on all Cattle services. If you have overridden the settings using environment varibles, Java system properties, or local config files, the changes to the database will not matter as the configuration is not coming from the database. Refer to the "source" property in the API to determine from where the configuration is being read.

5.1.1 Environment Variables

Environment variables have a special format. Configuration keys in Cattle are dot separated names like `db.cattle.database`. Environment variables can not have dots in their name so `.` is replaced by _. Additionally, in order to not read all the environment variables available and only read the environment variables specific to Cattle, the environment variable must start with `CATTLE_`. The `CATTLE_` prefix will be stripped. To sum this all up, `db.cattle.database`, if set through an environment variable, must be `CATTLE_DB_CATTLE_DATABASE`.

5.2 Cattle Home

Extensions, logs, and the embedded database are by default stored in `$CATTLE_HOME`. Also, `$CATTLE_HOME/etc/cattle` is automatically added to the classpath, so `$CATTLE_HOME/etc/cattle/cattle.properties` will be found as the cattle config file and read.

If you are running the `cattle.jar` manually from the command line, `$CATTLE_HOME` defaults to `$HOME/.cattle`.

If you are running Cattle from the docker image "cattle/server," the `$CATTLE_HOME` is by default `/var/lib/cattle`. If you want to customize Cattle while running docker you have two options (and plenty more you could think of).

1. **Build a custom image**: You can always build a custom image that packages your contents of `$CATTLE_HOME`, for example:

   ```
   FROM cattle/server
   ADD cattle-home /var/lib/cattle
   ```

2. **Map a local folder to /var/lib/cattle**: You can also just map a local folder to `/var/lib/cattle`, for example: `docker run -v $(pwd)/cattle-home:/var/lib/cattle cattle/server`

CHAPTER 6

Examples

Contents:

6.1 Simple External Process Handler

To add logic to an orchestration process you must implement a Process Handler. You can write Java code and package it with Cattle, or the more desirable approach is to write the logic in the language of your choice and run that externally. This example uses bash scripting to integrate with the "instance.start" process. Refer to the inline comments in https://github.com/cattleio/cattle/blob/master/docs/examples/handler-bash/simple_handler.sh

6.2 Adding a hypervisor

Add a new hypervisor is relatively straight forward. You just need to subscribe to and handle 7 agent events. Refer to the inline comments in https://github.com/cattleio/cattle/blob/master/docs/examples/handler-bash/hypervisor.sh. The events you must handle are as follows:

storage.image.activate: Download and make the image ready for use on this hypervisor.

storage.image.remove: Remove the specified image. All volumes associated to this image will have already been deleted from the hypervisor.

storage.volume.activate: Create or enable the volume. If the volume already exists, perform any other actions that may be necessary right before an instance starts using the volume.

storage.volume.deactivate: Called after an instance is shutdown and the volume is not in use anymore. The volume should not be deleted, just deactivated. For many volume formats there is nothing to do for deactivate.

storage.volume.remove: Remove the specified volume.

compute.instance.activate: Turn on the instance (virtual machine or container). You can assume that the storage is already created and available.

compute.instance.deactivate: Turn off the instance.

6.3 Customizing Libvirt Configuration

Launching a virtual machine using libvirt is largely determined by the domain XML that is passed to libvirt. The domain XML is created using Mako templates to make it very easy to tweak. The default libvirt XML template is setup with blocks designed to be customized using Mako's inheritance. This way you can easily package and use your own custom libvirt domain XML template that overrides a section of the configuration and adds items specific to your setup. Refer to https://github.com/cattleio/cattle/tree/master/docs/examples/libvirt/hostfs for more information.