By: Justin Ellingwood

Subscribe

20

How To Use Fleet and Fleetctl to Manage your CoreOS Cluster

Posted Sep 12, 2014 42.7k Clustering Scaling Networking System Tools CoreOS

Tutorial Series

This tutorial is part 3 of 9 in the series: Getting Started with CoreOS

Introduction

CoreOS provides an excellent environment for managing Docker containers across multi-server environments. One of the most essential components for making this cluster management simple is a service called **fleet**.

Fleet allows users to manage Docker containers as services for their entire cluster. It works by acting as an interface and a level of abstraction over each cluster member's systemd init system. Users can set constraints that affect the conditions under which a service runs. This lets administrators define what they would like their infrastructure to look like by telling certain applications to run on the same or separate hosts based on the supplied criteria.

In this guide, we will explore fleet and the `fleetctl` utility that allows you to control the daemon.

Prerequisites

To follow along with this guide, you should have a CoreOS cluster available.

The cluster we are using in this guide can be created by following our guide on how to create a CoreOS cluster on DigitalOcean. We will assume that you have the cluster configuration described in that guide.

The cluster that has been configured has three nodes. They are configured to communicate between each other using the private networking interface. The public interface is available on each of these nodes for running public services. The node names are:

- coreos-1
- coreos-2
- coreos-3

When you have your cluster ready, continue on to learn more about fleet.

Working with Service Unit Files

Before we get into the `fleetctl` tool, we should talk a bit about service unit files.

Unit files are used by the `systemd` init system to describe each available service, define the commands needed to manage it, and set dependency information to ensure the system is in a workable state when each service is started. The `fleet` daemon is built on top of `systemd` in order to manage services on a cluster-wide level. Because of this, it uses slightly modified versions of standard `systemd` unit files.

To get an in-depth look at `fleet` unit files, follow our deep dive on the subject. For this guide, we will just be giving a rough overview of the format of these files. We will also provide an example unit file that you can use to learn about `fleetctl`.

Fleet Unit File Sections

The basic sections that most unit files will have are:

- **Unit**: This section is used to provide generic information about the unit that is not dependent on the "type" of unit. This will include metadata information and dependency information. This is mainly used in `fleet` to provide descriptions and specify this units place in connection to other service units.
- **Unit Type Section**: The `fleet` daemon can take units of different types, including:
  - Service
  - Socket
  - Device
  - Mount
  - Automount
  - Timer
  - Path

If the type has specific options, then a section of the associated type is allowed. The `Service` section type is by far the most common. This section is used to define type-specific attributes. For `Service` units, this usually involves defining the start and stop commands, as well as the pre and post start or stop commands that might perform associated actions.

- **X-Fleet**: This section is used to provide fleet-specific configuration options. This mainly means that you can specify that a service must or must not be scheduled in a certain way based on criteria like machine ID, currently running services, metadata information, etc.

The general format of a unit file will be:

```
[Unit]
Generic_option_1
Generic_option_2

[Service]
Service_specific_option_1
Service_specific_option_2

[X-Fleet]
Fleet_option_1
Fleet_option_2
```

Sample Service Unit File

To get started with this tutorial, we will give you a unit file to use. This is taken from the CoreOS quickstart page as an example. On one of your CoreOS machines, type:

```
vim hello.service
```

Inside, type out our example service file:

```
[Unit]
```

```
Description=My Service
After=docker.service

[Service]
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill hello
ExecStartPre=-/usr/bin/docker rm hello
ExecStartPre=/usr/bin/docker pull busybox
ExecStart=/usr/bin/docker run --name hello busybox /bin/sh -c "while true; do echo Hello World; sleep 1;
done"
ExecStop=/usr/bin/docker stop hello
```

Let's quickly go over what this does.

In the `[Unit]` section, a description is set and we tell `systemd` that this service can only be run after the `docker.service` unit has started. This is because our unit relies on Docker to function.

In the `[Service]` section, we disable the starting timeout and then we set up some actions to run prior to starting the service. The `ExecStartPre` is executed before the main `ExecStart` action. If these are called with `=-` it means that the action can fail and not affect the service's completion. This is necessary since our pre-start actions basically tear down any previously running service that may have been running. This will fail if none can be found and we do not want that to stop our service from starting since this is just a cleanup procedure.

The last pre-start action pulls down the basic busybox image which will be used to run our commands. Since this is necessary, we don't use the `=-` syntax. We then start a container with this image with an infinite loop that prints out "Hello World" once a second. The stop action simply stops this container.

This will be just enough to get you up and running. If you want more information about how to develop fleet files, check out our guide on fleet unit files.

Basic Machine Management Commands

The first thing we will do is introduce you to the `fleetctl` utility. As a cluster administrator, this tool will be your main interface to manage your fleet of machines. Much of the syntax has been carried over from `systemctl`, systemd's management tool.

To start off, we can get a list of all of the cluster members by typing:

```
fleetctl list-machines
```

```
MACHINE     IP      METADATA
14ffe4c3... 10.132.249.212  -
1af37f7c... 10.132.249.206  -
9e389e93... 10.132.248.177  -
```

As you can see, each of your machines is listed here as being available. When each member bootstraps itself using the cloud-config file, it generates a unique machine ID, which is used to identify each node. This is written to a file at `/etc/machine-id`.

By default, fleet will use the machine's public IPv4 address for communication with other members. However, in our cloud-config file, we told fleet to use our private interfaces for communication. These are the IP addresses shown in the above output.

The "METADATA" column is currently blank in the above example. However, we could have added arbitrary key-value pairs under the `metadata` attribute for fleet in the cloud-config. This might look like this:

```
#cloud-config
. . .
coreos:
  fleet:
    public-ip: $private_ipv4
    metadata: region=europe,public_ip=$public_ipv4
```

If you set this in your cloud-config to bootstrap all of your machines, your output would look more like this:

```
MACHINE     IP      METADATA
14ffe4c3... 10.132.249.212  public_ip=104.131.36.200,region=europe
1af37f7c... 10.132.249.206  public_ip=104.131.15.192,region=europe
9e389e93... 10.132.248.177  public_ip=104.131.15.192,region=europe
```

This extra data is useful for quickly getting information about a node from a management perspective, but it can also be used in service definitions to target specific hosts.

To connect to a specific machine within the cluster, you can use the `fleetctl ssh` command. This will allow you to identify a machine to connect to based on its machine ID or machine associated with a supplied unit name.

For instance, if you have a unit running called `nginx.service`, you can connect to whatever host is running that service by typing:

```
fleetctl ssh nginx
```

You will be dropped into a shell session on the associated host. You can also run a single command on a remote host, just as you can when running with the normal `ssh` executable. For instance, to get the values of the `COREOS_PRIVATE_IPV4` and `COREOS_PUBLIC_IPV4` variables that CoreOS sets in a file called `/etc/environment` (based on the parameters of the cloud-config and the networking interfaces available), you can type:

```
fleetctl ssh nginx cat /etc/environment
```

```
COREOS_PRIVATE_IPV4=10.132.249.212
COREOS_PUBLIC_IPV4=104.131.29.80
```

Service Management

Most of the other commands available through `fleetctl` are based around service management.

Starting a Service

Starting a service involves quite a few steps. The service file must be uploaded into `fleet` so that it is aware of the unit. It must then be scheduled onto a specific machine out of the cluster. It can then be started. There are commands for each of these with `fleetctl`, with the commands responsible for the latter stages also performing the former if necessary.

You can use the `submit` command to submit your unit file into `fleet`. This will simply cause `fleet` to read the file contents into memory, making it available for further actions.

```
fleetctl submit hello.service
```

Your `hello.service` file is now known to `fleet`. To see the unit files that have been submitted, you can type:

```
fleetctl list-unit-files
```

```
UNIT        HASH    DSTATE      STATE       TMACHINE
hello.service   0d1c468 inactive    inactive    -
```

As you can see, the unit file is present, but has not been scheduled on any hosts or started.

To see the contents of a unit file that `fleet` knows about, you can type:

```
fleetctl cat hello.service
```

```
[Unit]
Description=My Service
After=docker.service

[Service]
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill hello
ExecStartPre=-/usr/bin/docker rm hello
ExecStartPre=/usr/bin/docker pull busybox
ExecStart=/usr/bin/docker run --name hello busybox /bin/sh -c "while true; do echo Hello World; sleep 1; done"
ExecStop=/usr/bin/docker stop hello
```

This will allow you to see the current file that `fleet` knows about.

Note: The submit command is idempotent, meaning that `fleet` will *not* update the in-memory unit file if you re-submit it. If you need to update your unit file, you must remove it completely and then re-submit it. We will cover how to do this later.

Once your unit is submitted, the next step is to schedule it on a machine. Scheduling the unit involves the `fleet` engine looking at the unit to decide on the best machine in the cluster to pass the unit to. This will be predicated on the conditions within the `[X-Fleet]` section of the unit, as well as the current work volume of each machine in the cluster. When the unit has been scheduled, it has been passed to the specific machine and loaded into the local `systemd` instance.

Use the `load` command to load and schedule the unit:

```
fleetctl load hello.service
```

```
Unit hello.service loaded on 14ffe4c3.../10.132.249.212
```

If you had not manually loaded your unit earlier, it will be loaded automatically as part of this process by searching for the appropriate filename in the current directory.

Now, if we check our unit files, we can see that it has been loaded. We can even see which machine it is scheduled on:

```
fleetctl list-unit-files
```

```
UNIT         HASH    DSTATE  STATE   TMACHINE
hello.service   0d1c468 loaded  loaded  14ffe4c3.../10.132.249.212
```

This is also our first opportunity to check out the `list-units` command. This command is used to show any running or scheduled units and their statuses:

```
fleetctl list-units
```

```
UNIT         MACHINE              ACTIVE     SUB
hello.service   14ffe4c3.../10.132.249.212  inactive    dead
```

To actually start a unit, you can use the `start` command. This will start the unit on the machine in which it has been loaded by executing the start commands defined in the unit file:

```
fleetctl start hello.service
```

```
Unit hello.service launched on 14ffe4c3.../10.132.249.212
```

Once again, we should check the `list-unit-files`:

```
fleetctl list-unit-files
```

```
UNIT         HASH    DSTATE      STATE      TMACHINE
hello.service   0d1c468 launched    launched    14ffe4c3.../10.132.249.212
```

In the above output, we can see that the service has been launched. The `DSTATE` column indicates the "desired state" and the `STATE` indicates the actual state. If these two match, this usually means that the action was successful.

We should also look at `list-units` again:

```
fleetctl list-units
```

```
UNIT         MACHINE              ACTIVE  SUB
hello.service   14ffe4c3.../10.132.249.212  active  running
```

This gives us information about the `systemd` state. It is collected directly from the local daemon, so this is a better picture of how the local system sees the service state. The `ACTIVE` column is a generalized state of the unit, while `SUB` is a more low-level description.

Stopping a Service

Each of the commands above have a companion command that reverses the state.

For instance, to stop a service from running, use the `stop` command. This will cause the local machine's `systemd` instance to execute the stopping commands defined in the unit:

```
fleetctl stop hello.service
```

```
Unit hello.service loaded on 14ffe4c3.../10.132.249.212
```

As you can see, the service has reverted back to the `loaded` state. This means that it is still loaded in the machine's `systemd`, but it is not currently running. We can confirm that here:

```
fleetctl list-unit-files
```

```
UNIT        HASH    DSTATE  STATE   TMACHINE
hello.service   0d1c468 loaded  loaded  14ffe4c3.../10.132.249.212
```

To remove the unit from that machine's systemd, but keep it available in `fleet`, you can `unload` the unit. If the unit is currently active, it will be stopped prior to being unloaded:

```
fleetctl unload hello.service
```

If we check the state, we can see that it is now marked as inactive. It also does not have a target machine listed:

```
fleetctl list-unit-files
```

```
UNIT        HASH    DSTATE      STATE       TMACHINE
hello.service   0d1c468 inactive    inactive    -
```

If we want to remove the unit from `fleet` entirely, we can use the `destroy` command. This will stop and unload the unit if necessary, and then remove the unit from `fleet`:

```
fleetctl destroy hello.service
```

If you modify a unit file, you *must* destroy the current unit in `fleet` before submitting/starting it again.

Getting Unit Statuses

You have already seen some of the methods of getting information about the status of units.

For instance, we have covered how `list-units` will list all of the units that have currently been scheduled on a machine:

```
fleetctl list-units
```

```
UNIT        MACHINE             ACTIVE  SUB
hello.service   14ffe4c3.../10.132.249.212  active  running
```

The `list-unit-files` provides a list of *all* units that `fleet` knows about. It also gives information about the desired and actual state:

```
fleetctl list-unit-files
```

```
UNIT        HASH    DSTATE      STATE       TMACHINE
hello.service   0d1c468 launched    launched    14ffe4c3.../10.132.249.212
```

For more specific information about a unit that has been started, there are a few other commands. The `status` command passes back the `systemctl status` result for the service on the host that is running the unit:

```
fleetctl status hello.service
```

```
● hello.service - My Service
   Loaded: loaded (/run/fleet/units/hello.service; linked-runtime)
   Active: active (running) since Mon 2014-09-08 21:51:22 UTC; 3min 57s ago
  Process: 7630 ExecStartPre=/usr/bin/docker pull busybox (code=exited, status=0/SUCCESS)
  Process: 7618 ExecStartPre=/usr/bin/docker rm hello (code=exited, status=0/SUCCESS)
  Process: 7609 ExecStartPre=/usr/bin/docker kill hello (code=exited, status=0/SUCCESS)
 Main PID: 7638 (docker)
   CGroup: /system.slice/hello.service
           └─7638 /usr/bin/docker run --name hello busybox /bin/sh -c while true; do echo Hello World; sleep
1; done

Sep 08 21:55:11 coreos-3 docker[7638]: Hello World
Sep 08 21:55:12 coreos-3 docker[7638]: Hello World
Sep 08 21:55:13 coreos-3 docker[7638]: Hello World
Sep 08 21:55:14 coreos-3 docker[7638]: Hello World
Sep 08 21:55:15 coreos-3 docker[7638]: Hello World
Sep 08 21:55:16 coreos-3 docker[7638]: Hello World
Sep 08 21:55:17 coreos-3 docker[7638]: Hello World
Sep 08 21:55:18 coreos-3 docker[7638]: Hello World
Sep 08 21:55:19 coreos-3 docker[7638]: Hello World
Sep 08 21:55:20 coreos-3 docker[7638]: Hello World
```

As you can see, we finally get to verify that the output of our unit is being produced.

Similarly, if you wish to see the journal entry for the service that is available on the associated machine, you can use the `journal` command:

```
fleetctl journal hello.service
```

```
-- Logs begin at Mon 2014-09-08 14:22:14 UTC, end at Mon 2014-09-08 21:55:47 UTC. --
Sep 08 21:55:38 coreos-3 docker[7638]: Hello World
Sep 08 21:55:39 coreos-3 docker[7638]: Hello World
Sep 08 21:55:40 coreos-3 docker[7638]: Hello World
Sep 08 21:55:41 coreos-3 docker[7638]: Hello World
Sep 08 21:55:42 coreos-3 docker[7638]: Hello World
Sep 08 21:55:43 coreos-3 docker[7638]: Hello World
Sep 08 21:55:44 coreos-3 docker[7638]: Hello World
Sep 08 21:55:45 coreos-3 docker[7638]: Hello World
Sep 08 21:55:46 coreos-3 docker[7638]: Hello World
Sep 08 21:55:47 coreos-3 docker[7638]: Hello World
```

By default, this will show the last 10 lines. You can adjust this by adding a `--lines` parameter, like this:

```
fleetctl journal --lines 20 hello.service
```

You can also use the `-f` parameter, which stands for "follow". This behaves in a similar way to `tail -f` in that it will continue to pass back the latest log entries:

```
fleetctl journal -f hello.service
```

Conclusion

By learning how to use `fleet` and `fleetctl` effectively, you can easily control your CoreOS cluster. Your services and containers can be moved around to different machines without much of a problem.

In a later guide, we'll discuss more in-depth how to create fleet unit files. This will allow you to create flexible and powerful services that takes advantage of the CoreOS architecture.

Author:
Justin Ellingwood

Tutorial Series

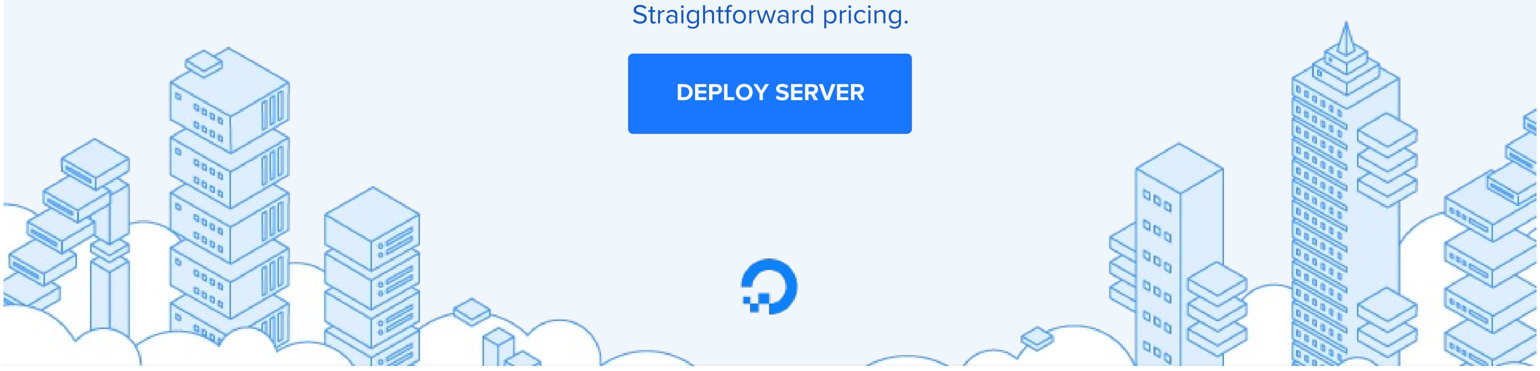
Getting Started with CoreOS

CoreOS is a powerful Linux distribution built to make large, scalable deployments on varied infrastructure simple to manage. Based on a build of Chrome OS, CoreOS maintains a lightweight host system and uses Docker containers for all applications. In this series, we will introduce you to the basics of CoreOS, teach you how to set up a CoreOS cluster, and get you started with using docker containers with CoreOS.

Spin up an SSD cloud server in under a minute.

Simple setup. Full root access.
Straightforward pricing.

DEPLOY SERVER



Related Tutorials

How To Deploy a Node.js and MongoDB Application with Rancher on Ubuntu 14.04

How To Create a Multi-Node MySQL Cluster on Ubuntu 16.04

How To Centralize Logs with Rsyslog, Logstash, and Elasticsearch on Ubuntu 14.04

How to Migrate Redis Data with Master-Slave Replication on Ubuntu 14.04

How To Use Ansible to Set Up a Production Elasticsearch Cluster

7 Comments

Leave a comment...

Logged in as:

Notify me of replies to my comment

Comment

stevem *September 17, 2014*

0

Great Demo.

I have run through this many times and receive the same error each time at the same point of the demo.

I am receiving the following error when I reach this command "fleetctl status hello.service" in the demo. The same error occurs with "fleetctl ssh" and "fleetctl journal"

"Error running remote command: SSH_ AUTH _SOCK environment variable is not set. Verify ssh-agent is running. See https://github.com/coreos/fleet/blob/master/Documentation/using-the-client.md for help."

Everything else is working well and the ssh-agent appears to be configured properly. What did I miss?

Thought I would post the error in case someone else is receiving the same error in the demo.

Thanks again for a great demo.

Steve

0

jellingwood MOD *September 18, 2014*

That usually means that you didn't correctly forward your user agent info when connecting to your CoreOS host.

On your home computer, start your agent by typing:

```
eval $(ssh-agent)
```

Then, add your private key to the agent by typing:

```
ssh-add
```

When you connect to your CoreOS host, pass the `-A` flag to forward your user agent info so that you can connect to the other cluster members from the one you are logged into:

```
ssh -A core@host
```

The commands should function correctly afterwards.

0

stevem *September 18, 2014*

Thx jellingwood. Worked perfect.

The magic in your answer was "home computer". It didn't occur to me that the problem could be local after a successful login and successfully running "fleetctl submit", "fleetctl load" and "fleetctl start" on the server.

The issue was local rather than on the server. Started from scratch using GitBash and generated new keys. Worked perfect. The -A fag was helpful as well.

Steve

0

jellingwood MOD *September 18, 2014*

@stevem: Awesome! Glad you got it sorted out!

0

gogasca *July 7, 2015*

I generated the private keys using **ssh-keygen -t rsa**, then use **ssh-add** <private-key.pem> command. After that I was able to connect using** ssh -A core@<public_ip>**

```
$ ssh-keygen -t rsa
```

```
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/gogasca/.ssh/id_rsa):
/Users/gogasca/Documents/OpenSource/Development/WebRTC/coreos.pem
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in
/Users/gogasca/Documents/OpenSource/Development/WebRTC/coreos.pem.
Your public key has been saved in
/Users/gogasca/Documents/OpenSource/Development/WebRTC/coreos.pem.pub.
```

```
$ ssh-add coreos.pem
Identity added: coreos.pem (coreos.pem)
```

nxqd.inbox *December 17, 2014*

0

I had this problem when exec the command `fleetctl list-machines` on coreos vagrant. Do you have any idea why ?

```
fleetctl list-machines
2014/12/17 07:38:07 INFO client.go:278: Failed getting response from http://127.0.0.1:4001/: dial tcp
127.0.0.1:4001: connection refused
2014/12/17 07:38:07 ERROR client.go:200: Unable to get result for {Get /_coreos.com/fleet/machines},
retrying in 100ms
2014/12/17 07:38:07 INFO client.go:278: Failed getting response from http://127.0.0.1:4001/: dial tcp
127.0.0.1:4001: connection refused
2014/12/17 07:38:07 ERROR client.go:200: Unable to get result for {Get /_coreos.com/fleet/machines},
retrying in 200ms
2014/12/17 07:38:07 INFO client.go:278: Failed getting response from http://127.0.0.1:4001/: dial tcp
127.0.0.1:4001: connection refused
2014/12/17 07:38:07 ERROR client.go:200: Unable to get result for {Get /_coreos.com/fleet/machines},
retrying in 400ms
2014/12/17 07:38:08 INFO client.go:278: Failed getting response from http://127.0.0.1:4001/: dial tcp
127.0.0.1:4001: connection refused
2014/12/17 07:38:08 ERROR client.go:200: Unable to get result for {Get /_coreos.com/fleet/machines},
retrying in 800ms
2014/12/17 07:38:09 INFO client.go:278: Failed getting response from http://127.0.0.1:4001/: dial tcp
127.0.0.1:4001: connection refused
2014/12/17 07:38:09 ERROR client.go:200: Unable to get result for {Get /_coreos.com/fleet/machines},
retrying in 1s
```

jellingwood MOD *December 17, 2014*

0

@nxqd.inbox: Hey!

It appears that there's an issue with `fleet` connecting with `etcd`. Try checking out this guide to troubleshoot. It looks like your problem may be covered by the "Checking the Individual Host" section.

cc BY NC SA

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

Copyright © 2016 DigitalOcean™ Inc.