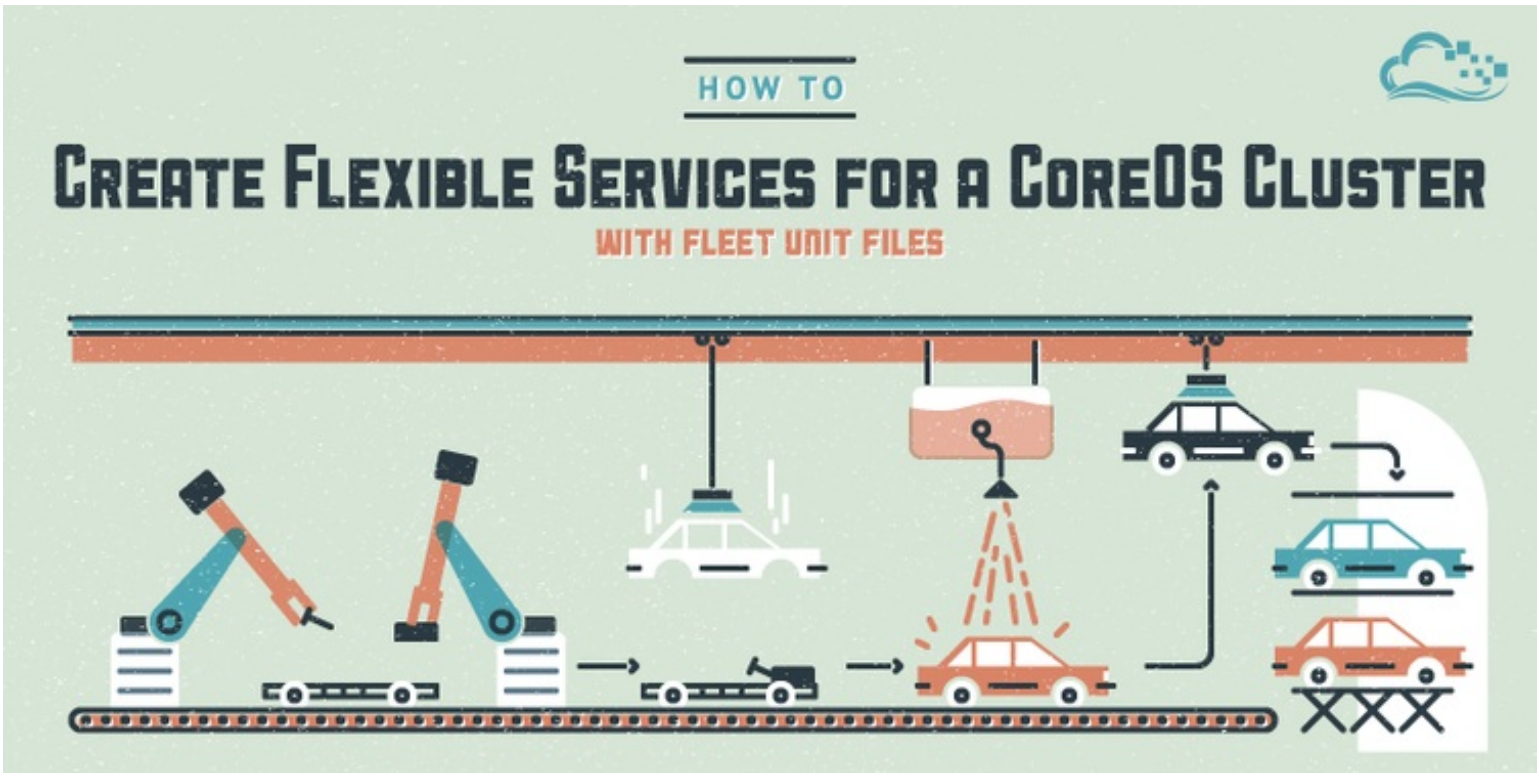
By: Justin Ellingwood

Subscribe



How to Create Flexible Services for a CoreOS Cluster with Fleet Unit Files

14

Posted Sep 12, 2014 43.3k Clustering Scaling Networking CoreOS

Tutorial Series

This tutorial is part 6 of 9 in the series: Getting Started with CoreOS

Introduction

CoreOS installations leverage a number of tools to make clustering and Docker contained services easy to manage. While `etcd` is involved in linking up the separate nodes and providing an area for global data, most of the actual service management and administration tasks involve working with the `fleet` daemon.

In a previous guide, we went over the basic usage of the `fleetctl` command for manipulating the services and cluster members. In that guide, we touched briefly on the unit files that fleet uses to define services, but these were simplified examples used to provide a working service to learn `fleetctl`.

In this guide, we will be exploring `fleet` unit files in depth to learn about how to create them and some techniques to make your services more robust in production.

Prerequisites

In order to complete this tutorial, we will assume that you have a CoreOS cluster configured as described in our clustering guide. This will leave you with three servers named as such:

- coreos-1

- coreos-2
- coreos-3

Although most of this tutorial will focus on unit file creation, these machines will be used later to demonstrate the scheduling affects of certain directives.

We will also assume that you have read our guide on how to use fleetctl. You should have a working knowledge of `fleetctl` so that you can submit and use these unit files with the cluster.

When you have completed these requirements, continue on with the rest of the guide.

Unit File Sections and Types

Because the service management aspect of `fleet` relies mainly on each local system's `systemd` init system, `systemd` unit files are used to define services.

While services are by far the most common unit type configured with CoreOS, there are actually other unit types that can be defined. These are a subset of those available to conventional `systemd` unit files. Each of these types are identified by the type being used as a file suffix, like `example.service`:

- **service**: This is the most common type of unit file. It is used to define a service or application that can be run on one of the machines in the cluster.
- **socket**: Defines details about a socket or socket-like files. These include network sockets, IPC sockets, and FIFO buffers. These are used to call services to start when traffic is seen on the file.
- **device**: Defines information about a device available in the udev device tree. Systemd will create these as needed on individual hosts for kernel devices based on udev rules. These are usually used for ordering issues to make sure devices are available before attempting to mount.
- **mount**: Defines information about a mount point for a device. These are named after the mount points they reference, with slashes replaced by dashes.
- **automount**: Defines an automount point. They follow the same naming convention as mount units and must be accompanied by the associated mount unit. These are used to describe on demand and parallelized mounting.
- **timer**: Defines a timer associated with another unit. When the point in time defined in this file is reached, the associated unit is started.
- **path**: Defines a path that can be monitored for path-based activation. This can be used to start another unit when changes are made to a certain path.

Although these options are all available, service units will be used most often. In this guide, we will only be discussing service unit configurations.

Unit files are simple text files ending with a dot and one of the above suffixes. Inside, they are organized by sections. For `fleet`, most unit files will have the following general format:

```
[Unit]
generic_unit_directive_1
generic_unit_directive_2

[Service]
service_specific_directive_1
service_specific_directive_2
service_specific_directive_3

[X-Fleet]
```

```
fleet_specific_directive
```

The section headers and everything else in a unit file is case-sensitive. The `[Unit]` section is used to define generic information about a unit. Options that are common for all unit-types are generally placed here.

The `[Service]` section is used to set directives that are specific to service units. Most (but not all) of the unit-types above have associated sections for unit-type-specific information. Check out the generic systemd unit file man page for links to the different unit types to see more information.

The `[X-Fleet]` section is used to set scheduling requirements for the unit for use with `fleet`. Using this section, you can require that certain conditions be true in order for a unit to be scheduled on a host.

Building the Main Service

For this section, we will start with a variant of the unit file described in our basic guide on running services on CoreOS. The file is called `apache.1.service` and will look like this:

```
[Unit]
Description=Apache web server service

# Requirements
Requires=etcd.service
Requires=docker.service
Requires=apache-discovery.1.service

# Dependency ordering
After=etcd.service
After=docker.service
Before=apache-discovery.1.service

[Service]
# Let processes take awhile to start up (for first run Docker containers)
TimeoutStartSec=0

# Change killmode from "control-group" to "none" to let Docker remove
# work correctly.
KillMode=none

# Get CoreOS environmental variables
EnvironmentFile=/etc/environment

# Pre-start and Start
## Directives with "=-" are allowed to fail without consequence
ExecStartPre=-/usr/bin/docker kill apache
ExecStartPre=-/usr/bin/docker rm apache
ExecStartPre=/usr/bin/docker pull username/apache
ExecStart=/usr/bin/docker run --name apache -p ${COREOS_PUBLIC_IPV4}:80:80 \
username/apache /usr/sbin/apache2ctl -D FOREGROUND

# Stop
ExecStop=/usr/bin/docker stop apache

[X-Fleet]
# Don't schedule on the same machine as other Apache instances
X-Conflicts=apache.*.service
```

We start with the `[Unit]` section. Here, the basic idea is to describe the unit and lay down the dependency information. We start

with a set of requirements. We have used hard requirements for this example. If we wanted `fleet` to attempt to start additional services, but not stop on a failure, we could have used the `Wants` directive instead.

Afterwards, we explicitly list out what the ordering of the requirements should be. This is important so that the prerequisite services are available when they are needed. It is also the way that we automatically kick off the sidekick etcd announce service that we will be building.

For the `[Service]` section, we turn off the service startup timeout. The first time a service is run on a host, the container will have to be pulled down from the Docker registry, which counts towards the startup timeout. This is defaulted to 90 seconds, which will typically enough time, but with more complex containers, it can take longer.

We then set the killmode to none. This is used because the normal kill mode (control-group) will sometimes cause container removal commands to fail (especially when attempted by Docker's `--rm` option). This can cause issues on next restart.

We pull in the environment file so that we have access to the `COREOS_PUBLIC_IPV4` and, if private networking was enabled during creation, the `COREOS_PRIVATE_IPV4` environmental variables. These are very useful for configuring Docker containers to use their specific host's information.

The `ExecStartPre` lines are used to tear down any leftover cruft from previous runs to make sure the execution environment is clean. We use `=-` on the first two of these to indicate that `systemd` should ignore and continue if these commands fail. Because of this, Docker will attempt to kill and remove previous containers, but will not worry if it cannot find any. The last pre-start is used to ensure that the most up-to-date version of the container is being run.

The actual start command boots the Docker container and binds it to the host machine's public IPv4 interface. This uses the info in the environment file and makes it trivial to switch interfaces and ports. The process is run in the foreground because the container will exit if the running process ends. The stop command attempts to stop the container gracefully.

The `[X-Fleet]` section contains a simple condition that forces `fleet` to schedule the service on a machine that is not already running another Apache service. This is an easy way to make a service highly available by forcing duplicate services to start on separate machines.

Basic Take-Aways For Building Main Services

In the above example, we went over a fairly basic configuration. However, there are plenty of lessons that we can can from this to assist us in building services in general.

Some behavior to keep in mind when building a main service:

- **Separate logic for dependencies and ordering**: Lay out your dependencies with `Requires=` or `Wants=` directives dependent on whether the unit you are building should fail if the dependency cannot be fulfilled. Separate out the ordering with separate `After=` and `Before=` lines so that you can easily adjust if the requirements change. Separating the dependency list from ordering can help you debug in case of dependency issues.
- **Handle service registration with a separate process**: Your service should be registered with `etcd` to take advantage of service discovery and the dynamic configuration features that this allows. *However*, this should be handled by a separate "sidekick" container to keep the logic separate. This will allow you to more accurately report on the service's health as seen from an outside perspective, which is what other components will need.
- **Be aware of the possibility of your service timing out**: Consider adjusting the `TimeoutStartSec` directive in order to allow for longer start times. Setting this to "0" will disable a startup timeout. This is often necessary because there are times when Docker has to pull an image (on first-run or when updates are found), which can add significant time to initializing the service.
- **Adjust the KillMode if your service does not stop cleanly**: Be aware of the `KillMode` option if your services or containers seem to be stopping uncleanly. Setting this to "none" can sometimes resolve issues with your containers not being removed after stopping. This is especially important when you name your containers since Docker will fail if a container with the same

name has been left behind from a previous run. Check out the documentation on KillMode for more information

- **Clean up the environment before starting up**: Related to the above item, make sure to clean up previous Docker containers at each start. You should not assume that the previous run of the service exited as expected. These cleanup lines should use the `=-` specifier to allow them to fail silently if no cleanup is needed. While you should stop containers with `docker stop` normally, you should probably use `docker kill` during cleanup.
- **Pull in and use host-specific information for service portability**: If you need to bind your service to a specific network interface, pull in the `/etc/environment` file to get access to `COREOS_PUBLIC_IPV4` and, if configured, `COREOS_PRIVATE_IPV4`. If you need to know the hostname of the machine running your service, use the `%H` systemd specifier. To learn more about possible specifiers, check out the systemd specifiers docs. In the `[X-Fleet]` section, only the `%n`, `%N`, `%i`, and `%p` specifiers will work.

Building the Sidekick Announce Service

Now that we have a good idea about what to keep in mind when building a main service, we can get started looking at a conventional "sidekick" service. These sidekick services are associated with a main service and are used as an external point to register services with `etcd`.

This file, as it was referenced in the main unit file, is called `apache-discovery.1.service` and looks like this:

```
[Unit]
Description=Apache web server etcd registration

# Requirements
Requires=etcd.service
Requires=apache.1.service

# Dependency ordering and binding
After=etcd.service
After=apache.1.service
BindsTo=apache.1.service

[Service]

# Get CoreOS environmental variables
EnvironmentFile=/etc/environment

# Start
## Test whether service is accessible and then register useful information
ExecStart=/bin/bash -c '\
  while true; do \
    curl -f ${COREOS_PUBLIC_IPV4}:80; \
    if [ $? -eq 0 ]; then \
      etcdctl set /services/apache/${COREOS_PUBLIC_IPV4} \'{"host": "%H", "ipv4_addr":
${COREOS_PUBLIC_IPV4}, "port": 80}\' --ttl 30; \
    else \
      etcdctl rm /services/apache/${COREOS_PUBLIC_IPV4}; \
    fi; \
    sleep 20; \
  done'

# Stop
ExecStop=/usr/bin/etcdctl rm /services/apache/${COREOS_PUBLIC_IPV4}

[X-Fleet]
# Schedule on the same machine as the associated Apache service
X-ConditionMachineOf=apache.1.service
```

We start the sidekick service off in much the same way as we did the main service. We describe the purpose of the unit before

moving on to dependency information and ordering logic.

The first new item here is the `BindsTo=` directive. This directive causes this unit to follow the start, stop, and restart commands sent to the listed unit. Basically, this means that we can manage both of these units by manipulating the main unit once both are loaded into `fleet`. This is a one-way mechanism, so controlling the sidekick will not affect the main unit.

For the `[Service]` section, we again source the `/etc/environment` file because we need the variables it holds. The `ExecStart=` directive in this instance is basically a short `bash` script. It attempts to connect to the main services using the interface and port that was exposed.

If the connection is successful, the `etcdctl` command is used to set a key at the host machine's public IP address within `/services/apache` within `etcd`. The value of this is a JSON object containing information about the service. The key is set to expire in 30 seconds so if this unit goes down unexpectedly, stale service information won't be left in `etcd`. If the connection fails, the key is removed immediately since the service cannot be verified to be available.

This loop includes a 20 second sleep command. This means that every 20 seconds (before the 30 second `etcd` key timeout), this unit re-checks whether the main unit is available and resets the key. This basically refreshes the TTL on the key so that it will be considered valid for another 30 seconds.

The stop command in this case just results in a manual removal of the key. This will cause the service registration to be removed when the main unit's stop command is mirrored to this unit due to the `BindsTo=` directive.

For the `[X-Fleet]` section, we need to ensure that this unit is started on the same server as the main unit. While this does not allow the unit to report on the availability of the service to remote machines, it is important for the `BindsTo=` directive to function correctly.

Basic Take-Aways For Building Sidekick Services

In building this sidekick, we can see some things that we should keep in mind as a general rule for these types of units:

- **Check the actual availability of the main unit**: It is important to actually check the state of main unit. Do not assume that the main unit is available just because the sidekick has been initialized. This is dependent on what the main unit's design and functionality is, but the more robust your check, the more credible your registration state will be. The check can be anything that makes sense for the unit, from checking a `/health` endpoint to attempting to connect to a database with a client.
- **Loop the registration logic to re-check regularly**: Checking the availability of the service at start is important, but it is also essential that you recheck at regular intervals. This can catch instances of unexpected service failures, especially if they somehow result in the container not stopping. The pause between cycles will have to be tweaked according to your needs by weighing the importance of quick discovery against the additional load on your main unit.
- **Use the TTL flag when registering with etcd for automatic de-registration on failures**: Unexpected failures of the sidekick unit can result in stale discovery information in `etcd`. To avoid conflicts between the registered and actual state of your services, you should let your keys time out. With the looping construct above, you can refresh each key before the timeout interval to make sure that the key never actually expires while the sidekick is running. The sleep interval in your loop should be set to slightly less than your timeout interval to ensure this functions correctly.
- **Register useful information with etcd, not just a confirmation**: During your first iteration of a sidekick, you may only be interested in accurately registering with `etcd` when the unit is started. However, this is a missed opportunity to provide plenty of useful information for other services to utilize. While you may not need this information now, it will become more useful as you build other components with the ability to read values from `etcd` for their own configurations. The `etcd` service is a global key-value store, so don't forget to leverage this by providing key information. Storing details in JSON objects is a good way to pass multiple pieces of information.

By keeping these considerations in mind, you can begin to build robust registration units that will be able to intelligently ensure that `etcd` has the correct information.

Fleet-Specific Considerations

While `fleet` unit files are for the most part no different from conventional `systemd` unit files, there are some additional capabilities and pitfalls.

The most obvious difference is the addition of a section called `[X-Fleet]` which can be used to direct `fleet` on how to make scheduling decisions. The available options are:

- **X-ConditionMachineID**: This can be used to specify an exact machine to load the unit. The provided value is a complete machine ID. This value can be retrieved from an individual member of the cluster by examining the `/etc/machine-id` file, or through `fleetctl` by issuing the `list-machines -l` command. The entire ID string is required. This may be needed if you are running a database with the data directory kept on a specific machine. Unless you have a specific reason to use this, try to avoid it because it decreases the flexibility of the unit.
- **X-ConditionMachineOf**: This directive can be used to schedule this unit on the same machine that is loaded with the specified unit. This is helpful for sidekick units or for lumping associated units together.
- **X-Conflicts**: This is the opposite of the above declaration, in that it specifies unit files which this unit *cannot* be scheduled alongside. This is useful for easily configuring high availability by starting multiple versions of the same service, each on a different machine.
- **X-ConditionMachineMetadata**: This is used to specify scheduling requirements based on the metadata of the machines available. In the "METADATA" column of the `fleetctl list-machines` output, you can see the metadata that has been set for each host. To set metadata, pass it in your `cloud-config` file when initializing the server instance.
- **Global**: This is a special directive that takes a boolean argument indicating whether this should be scheduled on all machines in the cluster. Only the metadata conditional can be used alongside this directive.

These additional directives give an administrator greater flexibility and power in defining how services should be run on the machines available. These are evaluated prior to passing them to a specific machine's `systemd` instance during the `fleetctl load` stage.

This brings us to the next thing to be aware of when working with related units in `fleet`. The `fleetctl` utility does not evaluate dependency requirements outside of the `[X-Fleet]` section of the unit file. This leads to some interesting problems when working with companion units in `fleet`.

This means that, while the `fleetctl` tool will take the necessary step to get the target unit into the desired state, stepping through the submission, loading, and starting process as needed based on the command given, it will not do this for the dependencies of the unit.

So, if you have both your main and sidekick unit submitted, but not loaded, in `fleet`, typing `fleetctl start main.service` will load and then attempt to start the `main.service` unit. However, since the `sidekick.service` unit is not yet loaded, and because `fleetctl` will not evaluate the dependency information to bring the dependency units through the loading and starting process, the `main.service` unit will fail. This is because once the machine's `systemd` instance processes the `main.service` unit, it will not be able to find the `sidekick.service` when *it* evaluates the dependencies. The `sidekick.service` unit was never loaded on the machine.

To avoid this situation when dealing with companion units, you can start the services manually at the same time, not relying on the `BindsTo=` directive bringing the sidekick into a running state:

```
fleetctl start main.service sidekick.service
```

Another option is to ensure that the sidekick unit is at least loaded when the main unit is run. The loading stage is where a machine is selected and the unit file is submitted to local `systemd` instance. This will ensure that dependencies are satisfied and that the `BindsTo=` directive will be able to execute correctly to bring up the second unit:

```
fleetctl load main.service sidekick.service
fleetctl start main.service
```

Keep this in mind in the event that your related units are not responding correctly to your `fleetctl` commands.

Instances and Templates

One of the most powerful concepts when working with `fleet` is unit templates.

Unit templates rely on a feature of `systemd` called "instances". These are instantiated units that are created at runtime by processing a template unit file. The template file is for the most part very similar to a regular unit file, with a few small modifications. However, these are extremely powerful when utilized correctly.

Template files can be identified by the `@` in their filename. While a conventional service takes this form:

```
unit.service
```

A template file can look like this:

```
unit@.service
```

When a unit is instantiated from a template, its instance identifier is placed between the `@` and the `.service` suffix. This identifier is a unique string selected by the administrator:

```
unit@instance_id.service
```

The base unit name can be accessed from within the unit file by the `%p` specifier. Similarly, the given instance identifier can be accessed with `%i`.

Main Unit File as a Template

This means that instead of creating your main unit file called `apache.1.service` with the contents we saw earlier, you could create a template called `apache@.service` that looks like this:

```
[Unit]
Description=Apache web server service on port %i

# Requirements
Requires=etcd.service
Requires=docker.service
Requires=apache-discovery@%i.service

# Dependency ordering
After=etcd.service
After=docker.service
Before=apache-discovery@%i.service

[Service]
# Let processes take awhile to start up (for first run Docker containers)
TimeoutStartSec=0

# Change killmode from "control-group" to "none" to let Docker remove
# work correctly.
```

```
KillMode=none

# Get CoreOS environmental variables
EnvironmentFile=/etc/environment

# Pre-start and Start
## Directives with "=-" are allowed to fail without consequence
ExecStartPre=-/usr/bin/docker kill apache.%i
ExecStartPre=-/usr/bin/docker rm apache.%i
ExecStartPre=/usr/bin/docker pull username/apache
ExecStart=/usr/bin/docker run --name apache.%i -p ${COREOS_PUBLIC_IPV4}:%i:80 \
username/apache /usr/sbin/apache2ctl -D FOREGROUND

# Stop
ExecStop=/usr/bin/docker stop apache.%i

[X-Fleet]
# Don't schedule on the same machine as other Apache instances
X-Conflicts=apache@*.service
```

As you can see, we have modified the `apache-discovery.1.service` dependency to be `apache-discovery@%i.service`. This will mean that if we have an instance of this unit file called `apache@8888.service`, this will require a sidekick called `apache-discovery@8888.service`. The `%i` has been replaced by the instance identifier. In this case, we are using the identifier to hold dynamic information about the way our service is being run, specifically the port the Apache server will be available on.

To make this work, we are changing the `docker run` parameter that exposes the container's ports to a port on the host. In the static unit file, the parameter we used was `${COREOS_PUBLIC_IPV4}:80:80`, which mapped port 80 of the container to port 80 of the host on the public IPv4 interface. In this template file, we have replaced this with `${COREOS_PUBLIC_IPV4}:%i:80` since we are using the instance identifier to tell us what port to use. A clever choice for the instance identifier can mean greater flexibility within your template file.

The Docker name itself has also been modified so that it also uses a unique container name based on the instance ID. Keep in mind that Docker containers cannot use the `@` symbol, so we had choose a different name from the unit file. We modify all of the directives that operate on the Docker container.

In the `[X-Fleet]` section, we've also modified the scheduling information to recognize these instantiated units instead of the static kind we were using before.

Sidekick Unit as a Template

We can run through a similar procedure to adapt the our sidekick unit for templating.

Our new sidekick unit will be called `apache-discovery@.service` and will look like this:

```
[Unit]
Description=Apache web server on port %i etcd registration

# Requirements
Requires=etcd.service
Requires=apache@%i.service

# Dependency ordering and binding
After=etcd.service
After=apache@%i.service
BindsTo=apache@%i.service
```

```
[Service]

# Get CoreOS environmental variables
EnvironmentFile=/etc/environment

# Start
## Test whether service is accessible and then register useful information
ExecStart=/bin/bash -c '\
  while true; do \
    curl -f ${COREOS_PUBLIC_IPV4}:%i; \
    if [ $? -eq 0 ]; then \
      etcdctl set /services/apache/${COREOS_PUBLIC_IPV4} \'{"host": "%H", "ipv4_addr":
${COREOS_PUBLIC_IPV4}, "port": %i}\' --ttl 30; \
    else \
      etcdctl rm /services/apache/${COREOS_PUBLIC_IPV4}; \
    fi; \
    sleep 20; \
  done'

# Stop
ExecStop=/usr/bin/etcdctl rm /services/apache/${COREOS_PUBLIC_IPV4}

[X-Fleet]
# Schedule on the same machine as the associated Apache service
X-ConditionMachineOf=apache@%i.service
```

We have gone through the same steps of requiring and binding to the instantiated version of the main unit processes instead of the static version. This will match the instantiated sidekick unit with the correct instantiated main unit.

During the `curl` command, when we are checking the actual availability of the service, we replace the static port 80 with the instant ID so that it is connecting to the correct place. This is necessary since we changed the port exposure mapping within the Docker command for our main unit.

We also modify the "port" being logged to `etcd` so that it uses this same instance ID. With this change, the JSON data being set in `etcd` is entirely dynamic. It will pick up the hostname, the IP address, and the port where the service is being run.

Finally, we change the conditional in the `[X-Fleet]` section again. We need to make sure this process is started on the same machine as the main unit instance.

Instantiating Units from Templates

To actually instantiate units from a template file, you have a few different options.

Both `fleet` and `systemd` can handle symbolic links, which gives us the option of creating links with the full instance IDs to the template files, like this:

```
ln -s apache@.service apache@8888.service
ln -s apache-discovery@.service apache-discovery@8888.service
```

This will create two links, called `apache@8888.service` and `apache-discovery@8888.service`. Each of these have all of the information needed for `fleet` and `systemd` to run these units now. However, they are pointed back to the templates so that we can make any changes needed in a single place.

We can then submit, load, or start these services with `fleetctl` like this:

```
fleetctl start apache@8888.service apache-discovery@8888.service
```

If you do not want to make symbolic links to define your instances, another option is to submit the templates themselves into `fleetctl`, like this:

```
fleetctl submit apache@.service apache-discovery@.service
```

You can instantiate units from these templates right from within `fleetctl` by just assigning instance identifiers at runtime. For instance, you could get the same service running by typing:

```
fleetctl start apache@8888.service apache-discovery@8888.service
```

This eliminates the need for symbolic links. Some administrators prefer the linking mechanism though since it means that you have the instance files available at any time. It also allows you to pass in a directory to `fleetctl` so that everything is started at once.

For instance, in your working directory, you may have a subdirectory called `templates` for your template files and a subdirectory called `instances` for the instantiated linked versions. You could even have one called `static` for non-templated units. You could do this like so:

```
mkdir templates instances static
```

You could then move your static files into `static` and your template files into `templates`:

```
mv apache.1.service apache-discovery.1.service static
mv apache@.service apache-discovery@.service templates
```

From here, you can create the instance links you need. Let's run our service on ports `5555`, `6666`, and `7777`:

```
cd instances
ln -s ../templates/apache@.service apache@5555.service
ln -s ../templates/apache@.service apache@6666.service
ln -s ../templates/apache@.service apache@7777.service
ln -s ../templates/apache-discovery@.service apache-discovery@5555.service
ln -s ../templates/apache-discovery@.service apache-discovery@6666.service
ln -s ../templates/apache-discovery@.service apache-discovery@7777.service
```

You can then start all of your instances at once by typing something like:

```
cd ..
fleetctl start instances/*
```

This can be incredibly useful starting up your services quickly.

Conclusion

You should have a decent understanding of how to build unit files for `fleet` by this point. By taking advantage of some of the dynamic features available within unit files, you can make sure your services are evenly distributed, close to their dependencies, and registering useful information with `etcd`.

In a later guide, we will cover how to configure your containers to use the information you are registering with `etcd`. This can

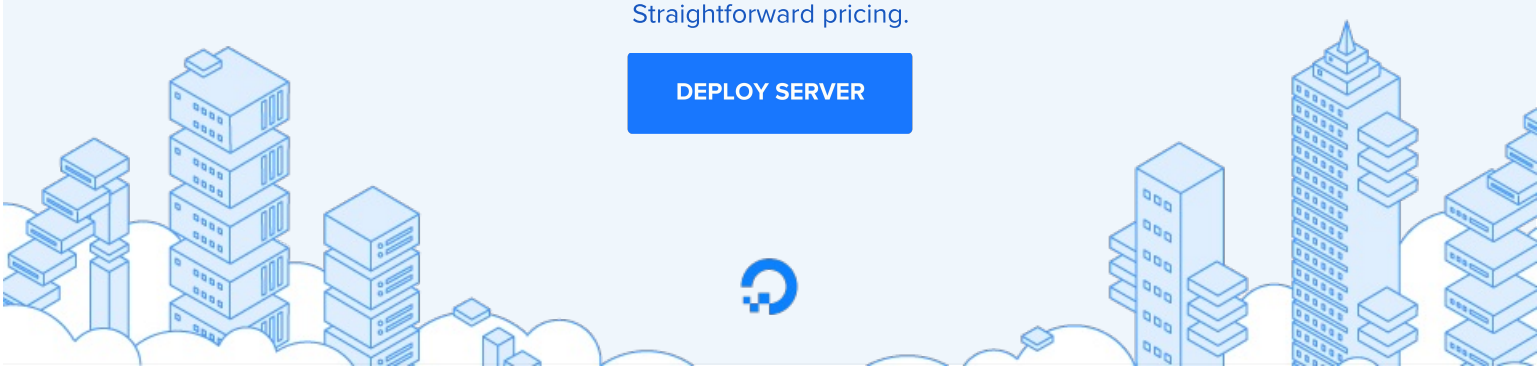
assist your services in building up a working knowledge of your actual deployment environment in order to pass requests to the appropriate containers in the backend.

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.



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

2 Comments

Leave a comment...

Logged in as:

Notify me of replies to my comment

Comment

flavius *September 21, 2014*

1

we will cover how to configure some of your configure your containers

jellingwood MOD *September 21, 2014*

0

Yikes, good catch. Should be all cleaned up now. Thanks!