

Docker 15

15.CoreOS 사용하기



CoreOS는 Docker 전용 리눅스 배포판입니다.

- Docker 컨테이너를 여러 서버에 손쉽게 배포
- 클러스터링
- 동적 확장
- 고가용성(High Availability)

위키백과

CoreOS는 응용 소프트웨어 배치, 보안, 신뢰성, 확장성의 용이함과 자동화에 초점을 맞춰 클러스터화된 전개용의 기반 시설을 제공하기 위한 리눅스 커널에 기반한 오픈 소스 경량 운영 체제이다.

CoreOS는 크게 etcd, systemd, fleet 세 가지 컴포넌트로 구성되어 있습니다.

ETCD

etcd는 분산 키-값(Distributed Key-Value) 저장소이며 클러스터의 설정 값과 노드 정보를 저장하고 공유하는 시스템입니다.

Apache Zookeeper, doozer와 비슷합니다.

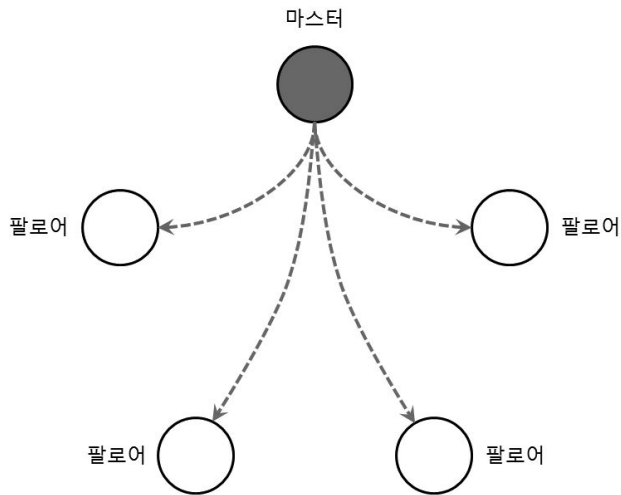
etcd는 다음과 같은 특징을 가지고 있습니다.

- HTTP 프로토콜에 JSON 형식 API를 제공합니다.
- 초당 1000회 쓰기 성능을 제공합니다.
- Raft 컨센서스 알고리즘을 이용하여 여러 서버들 중에서 마스터를 선출합니다.
- 키 자동 삭제 기능(TTL, Time to live)을 제공합니다.
- 언제나 데이터의 일관성을 보장하는 Atomic 읽기/쓰기를 제공합니다.
- HTTP 롱 폴링(long-polling)을 통해 키 변경 사항을 감시할 수 있습니다.

etcd는 모든 키 변경 사항을 로그로 저장합니다.

마스터는 로그를 각 팔로어에 복제하여 데이터를 공유합니다.

마스터뿐만 아니라 각 노드에서도 키를 추가하거나 값을 변경하면 모든 노드에 반영됩니다.



클러스터의 각 팔로어에 로그(데이터) 복제

systemd는 리눅스 서비스 매니저이며 기존 System V, BSD init 시스템을 대체하기 위해 개발되었습니다. CoreOS에서는 systemd를 통해 Docker 컨테이너를 실행합니다.

- 기존 init 시스템에 비해 부팅 속도가 매우 빠릅니다.
- 서비스 간의 의존성 관계를 설정할 수 있고, 실행 순서를 제어할 수 있습니다.
- 각 데몬의 로그는 journald를 사용하여 편리하게 조회할 수 있습니다.

systemd는 다음과 같은 형식의 유닛 파일로 서비스를 실행할 수 있습니다.
/etc/systemd/system 디렉터리 아래에 **example.service**와 같이 저장하면 됩니다.

SYSTEMD

/etc/systemd/system/example.service

```
[Unit]
Description=Example Service
Requires=docker.service
After=docker.service

[Service]
ExecStart=/usr/bin/docker run busybox /bin/sh -c "while true; do echo Hello World; sleep 1; done"

[Install]
WantedBy=multi-user.target
```

- **Unit:** 유닛 실행 설정입니다.
 - **Description:** 유닛 설명입니다.
 - **Requires:** 의존성 설정입니다. 예제는 서비스를 실행하려면 **docker.service**가 필요하다는 것을 뜻합니다.
 - **After:** 실행 순서 설정입니다. 예제에서는 **docker.service**가 완전히 실행된 뒤에 현재 서비스가 실행된다는 것을 뜻합니다.
- **Service:** 각 상황에 따라 실행할 명령을 설정합니다.
 - **ExecStart:** 현재 유닛이 시작되면 명령을 실행합니다. 예제에서는 **busybox** 이미지로 컨테이너를 생성한 뒤 셸에서 1초마다 **Hello World**를 출력합니다.
- **Install:** 프로세스를 실행할 **런 레벨**(타깃)을 설정합니다.
 - **WantedBy:** 유닛의 **런 레벨** 설정입니다. 예제에서는 **multi-user.target**으로 설정하여 **example.service** 파일이 /etc/systemd/multi-user.target.wants 디렉터리 아래에 링크됩니다.

런 레벨(Run Level)

런 레벨(Run Level)은 리눅스에서 운영체제의 실행 모드를 정의한 것입니다.

- 0, Halt: 시스템 종료입니다. 런 레벨 0으로 변경한다는 것은 시스템 종료를 뜻합니다 (runlevel0.target, poweroff.target).
- 1, Single-user Mode: 단일 사용자 모드입니다. 시스템 복원 모드라고도 합니다(runlevel1.target, rescue.target).
- 2, Multi-user Mode: 네트워크를 사용할 수 없는 다중 사용자 모드입니다(runlevel2.target, multi-user.target).
- 3, Multi-user Mode with Networking: 네트워크를 사용할 수 있는 일반적인 다중 사용자 모드입니다(runlevel3.target, multi-user.target).
- 4, User definable: 사용자 정의 모드(runlevel4.target, multi-user.target)
- 5, Multi-user graphical Mode: GUI 모드입니다(runlevel5.target, graphical.target).
- 6, Reboot: 재부팅입니다(runlevel6.target, reboot.target).

CoreOS에서는 GUI 모드가 없으며 주로 multi-user.target을 사용합니다

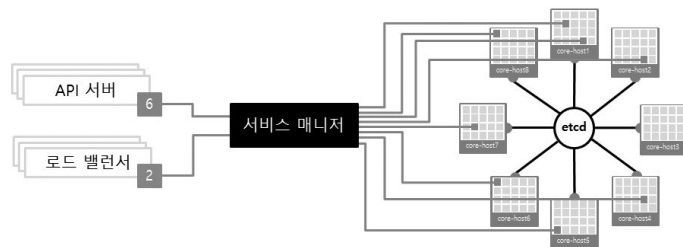
FLEET

fleet은 **etcd**와 **systemd**를 이용한 분산 서비스 실행(**init**) 시스템 입니다.
systemd는 로컬의 서비스를 관리하는 시스템이지만 **fleet**은 원격에서 여러 서버에 서비스를 실행할 수 있습니다.

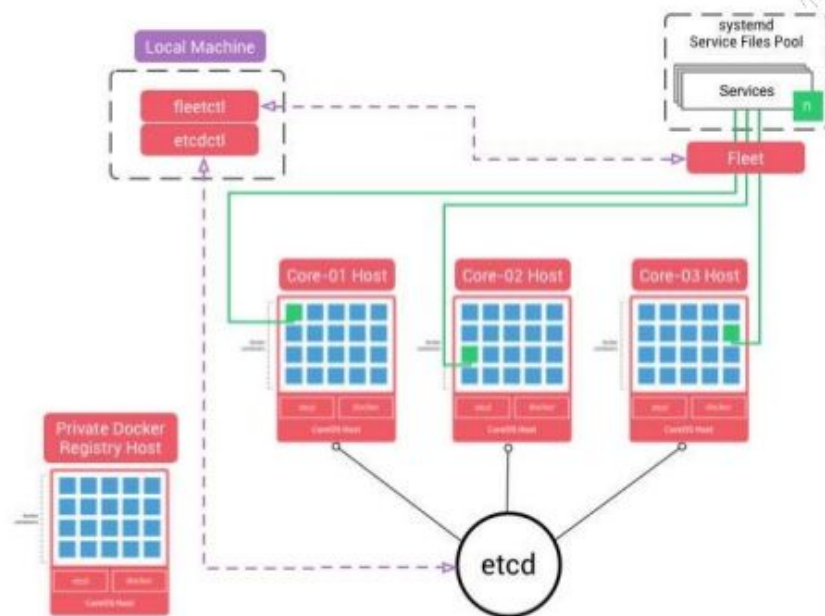
그림은 **fleet**를 이용하여 **API** 서버 컨테이너 6개와 로드 밸런서 컨테이너 2개를 **CoreOS** 클러스터에 배포한 모습입니다. 클러스터 상황에 맞게 **fleet**가 알아서 호스트를 선택한 뒤 **Docker** 컨테이너를 배포합니다. 또는, 특정 호스트를 설정하여 컨테이너를 배포할 수도 있습니다.

fleet의 특징입니다.

- **Docker** 컨테이너를 임의의 호스트에 배포합니다.
- 특정 호스트에 서비스가 물리지 않게 적절히 분산해줍니다.
- 특정 호스트에 장애가 발생해도 정해진 서비스 개수를 유지해줍니다. 즉 특정 호스트가 정지하면 해당 호스트에서 실행되던 서비스를 다른 호스트에서 실행해줍니다.
- 클러스터에 속한 호스트를 자동으로 발견(**discover**)합니다.



CoreOS Cluster Architecture

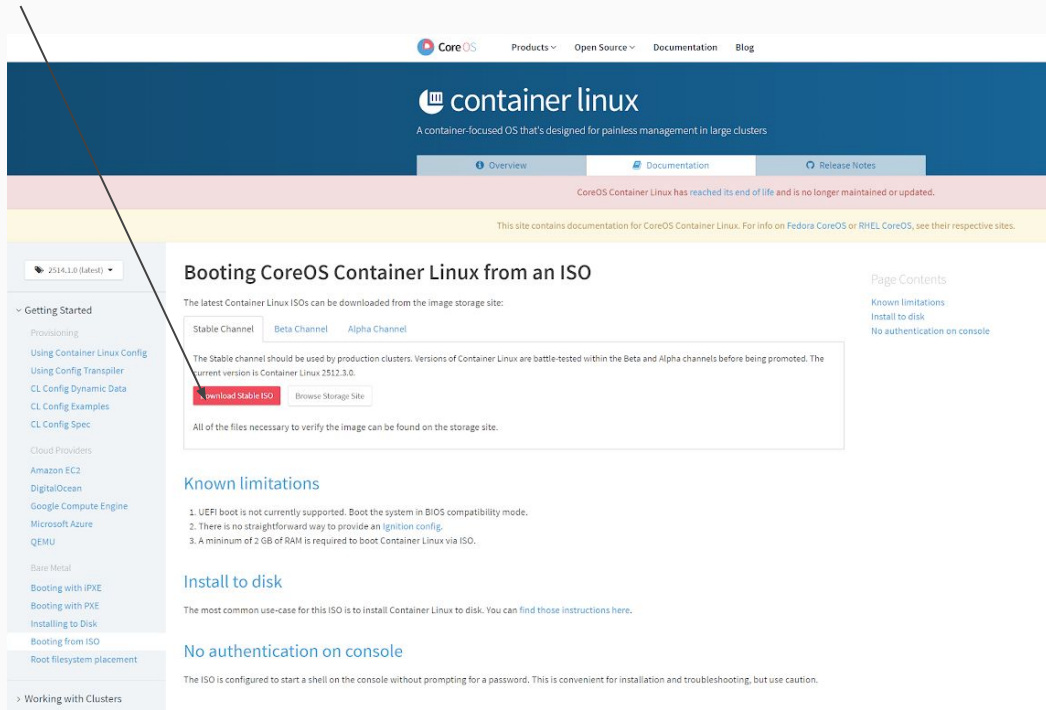


Source: <https://github.com/MarkMoudy/coreos-docker-ci-demo>

15-1.VirtualBox에 CoreOS설치하기

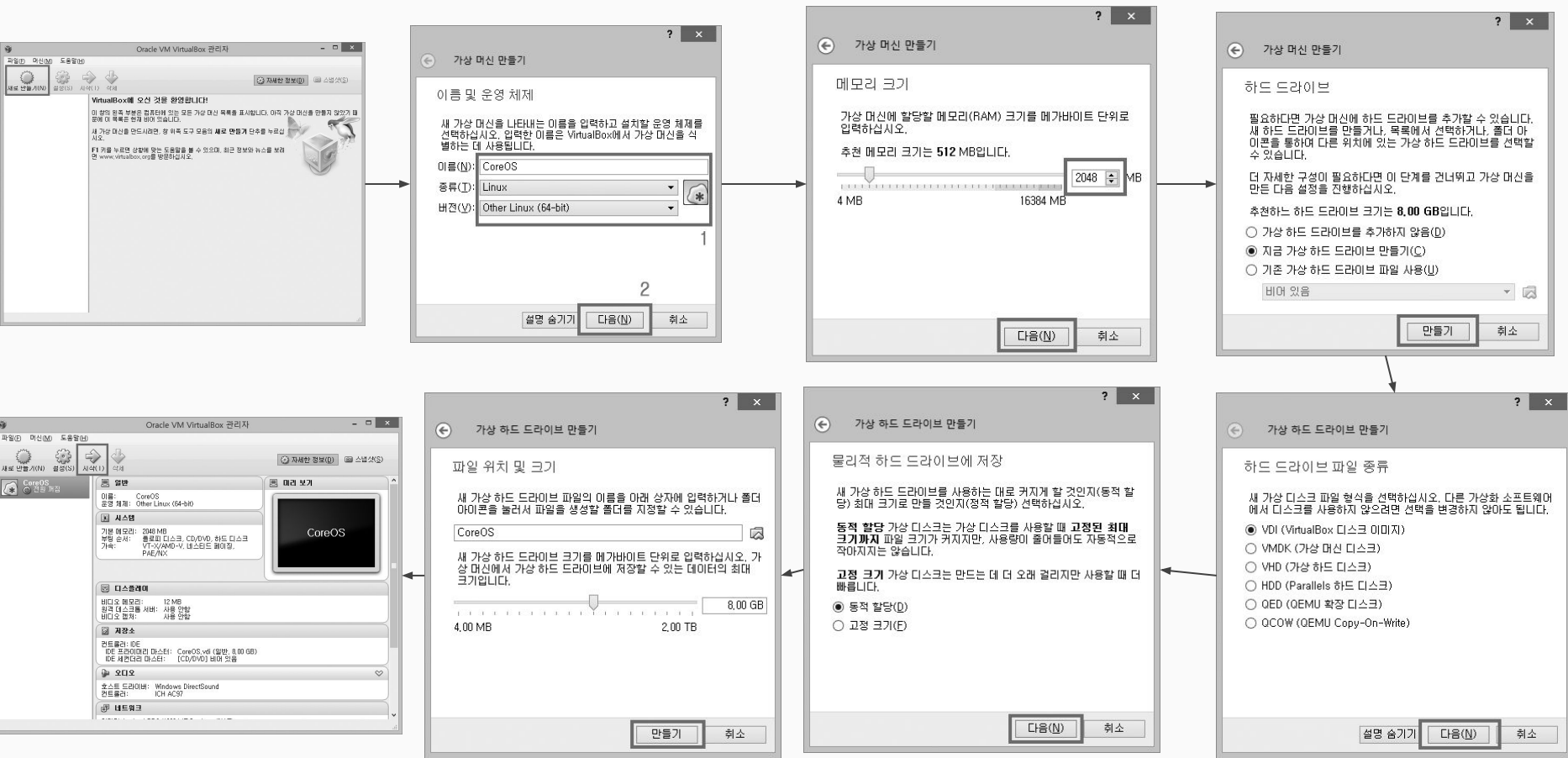
VirtualBox 설치 방법은 특별한 것이 없으므로 따로 설명하지 않습니다.

먼저 <https://coreos.com/docs/running-coreos/platforms/iso/>에 접속한 뒤 Download Stable ISO 버튼을 클릭하여 ISO 파일을 다운로드 합니다.

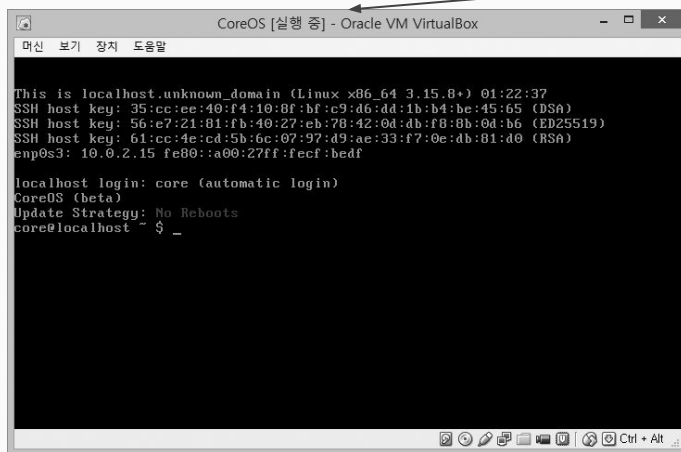
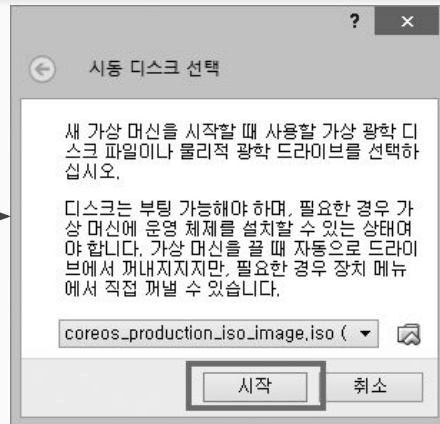
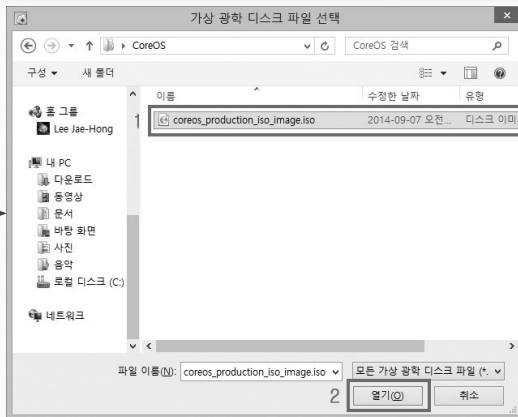
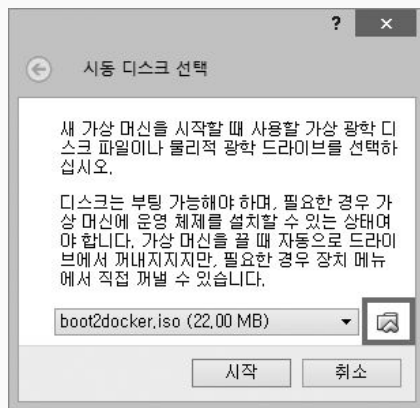


- **Stable Channel:** 많은 테스트를 거친 안정 버전입니다.
- **Beta Channel:** 알파 채널을 테스트한 뒤 등록된 버전입니다.
- **Alpha Channel:** 가장 최신 버전입니다. Docker, etcd, fleet 모두 최신 버전이 포함됩니다.

15-1.VirtualBox에 CoreOS설치하기



15-1.VirtualBox에 CoreOS설치하기



CoreOS ISO(CD) 파일로 부팅했습니다.
CoreOS는 따로 설치 화면이 없고 ISO 파일로 부팅한
`coreos-install` 명령을 사용하여 하드디스크에 설치하는
방식입니다.

15-1.VirtualBox에 CoreOS설치하기

간편하게 테스트하기 위해 SSH 키 설정 대신 비밀번호 방식을 사용하겠습니다.

VirtualBox 안에서 다음 명령을 입력하여 비밀번호를 생성한 뒤 **cloud-config.yaml**로 저장합니다.

```
$ openssl passwd -1 > cloud-config.yaml
Password: <사용할 비밀번호 입력>
Verifying - Password: <사용할 비밀번호 다시 입력>
```

이제 **vim**으로 **cloud-config.yaml** 파일을 열고 다음과 같이 작성합니다.

cloud-config.yaml

```
#cloud-config

users:
- name: exampleuser
  passwd: $1$qeijttft8$vs8v2Rnlw1iNkeAFnPWQ00
  groups:
    - sudo
    - docker
```

- **#cloud-config**는 반드시 첫째 줄에 입력합니다.
- **users:** 사용자 설정입니다.
 - **name:** 사용자 계정 이름입니다. 콘솔이나 SSH로 로그인할 때 이 사용자 이름을 사용하게 됩니다. **exampleuser**를 입력합니다.
 - **passwd:** 사용자 계정의 비밀번호 해시 값입니다. 앞에서 **openssl** 명령으로 생성한 비밀번호 해시 값을 그대로 사용합니다.
 - **group:** 사용자 계정의 그룹 설정입니다. root 권한으로 명령을 실행할 수 있도록 **sudo** 그룹을 설정하고, **docker**를 **sudo** 명령없이 실행할 수 있도록 **docker** 그룹을 설정합니다.

15-1.VirtualBox에 CoreOS설치하기

다음 명령을 실행하여 VirtualBox 가상 머신의 하드디스크(/dev/sda)에 CoreOS를 설치합니다.

```
$ sudo coreos-install -d /dev/sda -C stable -c cloud-config.yaml
```

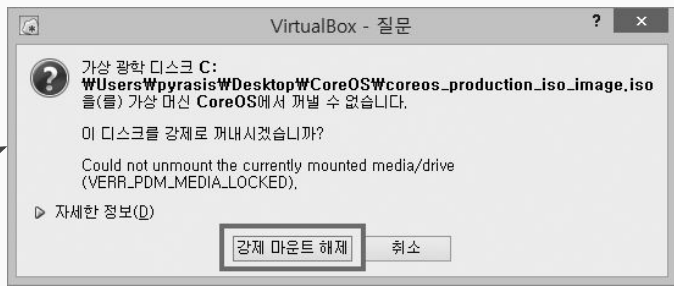
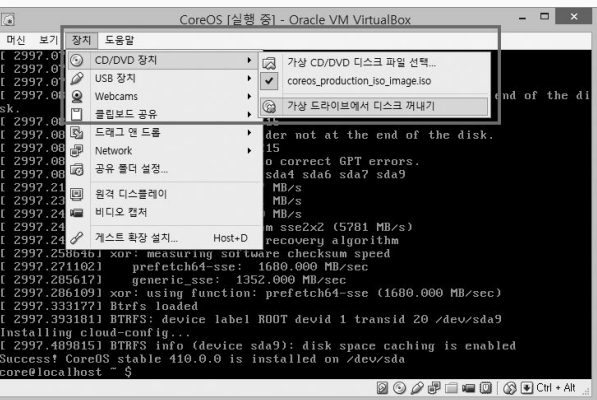
- -d: 설치할 하드디스크 장치입니다. **/dev/sda**를 설정합니다.
- -C: 릴리스 채널입니다. **stable**을 설정합니다. 알파, 베타 채널을 사용하려면 **alpha**, **beta**를 설정합니다.
- -c: 설정 파일 경로입니다. 앞에서 작성한 **cloud-config.yaml**을 설정합니다.

잠시 기다리면 CoreOS 설치가 완료됩니다.
다음과 같이 출력되면 설치가 끝난 것입니다.

```
Success! CoreOS stable <버전> is installed on /dev/sda
```

15-1.VirtualBox에 CoreOS설치하기

VirtualBox 가상 머신 화면에서 장치 → CD/DVD 장치 → 가상 드라이브에서 디스크 꺼내기를 클릭합니다



ISO 파일을 꺼냈으면 다음 명령을 입력하여 가상 머신을 재부팅합니다.

```
$ sudo reboot
```

VirtualBox 가상 머신이 재부팅된 뒤에 login에 **exampleuser**, Password에 앞에서 설정한 비밀번호를 입력하면 CoreOS에 로그인이 됩니다. VirtualBox 대신 SSH로 접속해도 됩니다.



15-1-1.systemd로 서비스 실행하기

/etc/systemd/system/example.service

```
[Unit]
Description=Example Service
Requires=docker.service
After=docker.service

[Service]
ExecStart=/usr/bin/docker run busybox /bin/sh -c "while true; do echo Hello World; sleep 1; done"

[Install]
WantedBy=multi-user.target
```

다음 내용을 /etc/systemd/system 디렉터리 아래 example.service 파일로 저장합니다.

```
$ sudo systemctl start example.service
```

이제 `systemctl` 명령으로 `example.service`를 실행합니다.
`systemctl start <systemd 유닛 이름>` 형식이며, 아무 에러가
출력되지 않았다면 정상적으로 서비스가 실행된 것입니다.

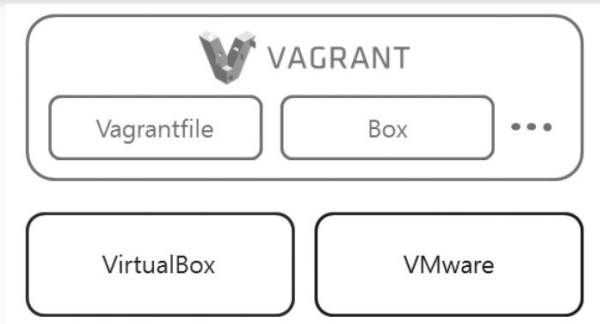
```
$ sudo journalctl -u example.service -f
```

`journalctl` 명령으로 Docker 컨테이너의 로그를 출력합니다.
`journalctl -u <systemd 유닛 이름>` 형식입니다. `-f` 옵션을
사용하면 로그를 실시간으로 출력합니다.

15-2.Vagrant로 CoreOS설치하기

Vagrant는 가상 머신을 손쉽게 생성하고 관리하는 도구입니다.

가상화 소프트웨어는 아니기 때문에 **VirtualBox**나 **VMware**가 반드시 설치되어 있어야 합니다.



Vagrant와 Docker

Vagrant도 Docker처럼 Vagrant Cloud(<https://vagrantcloud.com/>)를 통해 각종 리눅스 배포판 및 웹 서버, DB 등의 이미지(**Box**)를 제공하고 있습니다.

Vagrant는 가상화 소프트웨어인 **VirtualBox**와 **VMware**를 기반으로한 플랫폼이고,

Docker는 리눅스 커널의 **cgroups**, **namespaces**를 기반으로한 플랫폼입니다.

Vagrant Box는 Docker 이미지와는 달리 완전한 형태의 운영체제 형태입니다.















15-2.Vagrant로 CoreOS설치하기

앞에서 CoreOS를 설치할 때 터미널에서 일일이 설정 파일을 생성하고 설치 명령을 실행해야 했습니다. Vagrant를 이용하면 미리 설정된 환경을 이용하여 가상 머신 생성부터 운영체제 설치 및 설정까지 자동으로 진행할 수 있습니다.

다운로드 경로: <https://www.vagrantup.com/>

이제 Git으로 미리 작성된 CoreOS용 Vagrantfile을 받아옵니다

```
git clone https://github.com/coreos/coreos-vagrant.git
cd coreos-vagrant
```

 .gitattributes	Fixed typo in .gitattributes
 .gitignore	*: add support for ignition when using virtualbox
 CONTRIBUTING.md	doc: add CONTRIBUTING.md and MAINTAINERS
 DCO	chore(contributing): clean up CONTRIBUTING.md and split out DCO
 LICENSE	feat(*): initial commit
 MAINTAINERS	doc: add CONTRIBUTING.md and MAINTAINERS
 NOTICE	feat(*): initial commit
 README.md	vagrantfile: update for vagrant-ignition 0.0.2
 Vagrantfile	config: add support for \$update_channel
 cl.conf	*: add support for ignition when using virtualbox
 code-of-conduct.md	update CoC
 config.ign.sample	*: add support for ignition when using virtualbox
 config.rb.sample	config: add support for \$update_channel
 user-data.sample	Updated vagrant demo

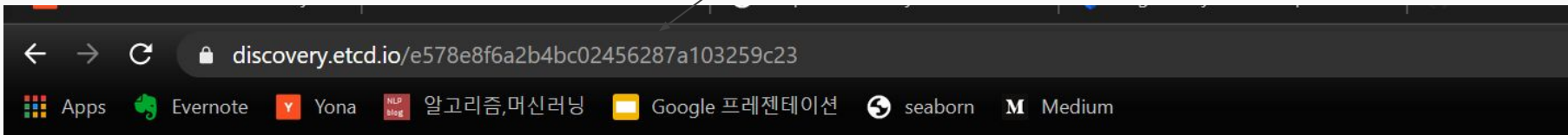
15-2.Vagrant로 CoreOS설치하기

이제 클러스터 상태를 저장하기 위해 **etcd discovery URL**을 할당받아야 합니다.
터미널에서 다음 명령을 실행하거나 웹 브라우저에서 <https://discovery.etcd.io/new>에 접속합니다.

```
$ curl -w "%n" https://discovery.etcd.io/new  
https://discovery.etcd.io/c0d96ba6f4024d5cabe2484e89bca52f
```



CoreOS에서는 **etcd**를 손쉽게 사용할 수 있도록 **discovery URL**을 서비스 형태로 제공합니다.
이 URL은 **etcd** 리더 노드(마스터)를 선출할 때 사용됩니다.



```
{"action": "get", "node": {"key": "/e578e8f6a2b4bc02456287a103259c23", "dir": true, "modifiedIndex": 34668012, "createdIndex": 34668012}}
```

15-2.Vagrant로 CoreOS설치하기

user-data

```
#cloud-config
coreos:
  etcd:
    discovery: https://discovery.etcd.io/<클러스터 ID>
    addr: $public_ipv4:4001
    peer-addr: $public_ipv4:7001
  fleet:
    public-ip: $public_ipv4
  units:
  - name: etcd.service
    command: start
  - name: fleet.service
    command: start
```

다음 내용을 **coreos-vagrant** 디렉터리 아래에 **user-data** 파일로 저장합니다.

- **#cloud-config**는 반드시 맨 앞 줄에 입력합니다.
- **coreos:** CoreOS 메인 설정입니다.
 - **discovery:** discovery URL입니다. 앞에서 할당받은 discovery URL을 설정합니다 (반드시 <https://discovery.etcd.io/>와 클러스터 ID를 포함하여 설정합니다).
 - **addr:** clientURL IP 주소이며 현재 서버의 IP 주소를 설정합니다. **\$public_ipv4**를 사용하면 Vagrant가 자동으로 가상 머신에 할당된 IP 주소를 설정해줍니다.
 - **peer-addr:** peerURL IP 주소이며 현재 서버의 IP 주소를 설정합니다. 마찬가지로 **\$public_ipv4**를 사용하면 Vagrant가 자동으로 가상 머신에 할당된 IP 주소를 설정해줍니다.
- **fleet:** fleet 설정입니다.
 - **public-ip:** 현재 서버의 IP 주소를 설정합니다. **\$public_ipv4**를 사용하면 Vagrant가 자동으로 가상 머신에 할당된 IP 주소를 설정해줍니다.
- **units:** 실행할 서비스 설정입니다.
 - **name:** 서비스 이름입니다. 여기서는 **etcd.service**, **fleet.service**를 실행합니다.
 - **command:** 서비스에 내릴 명령입니다. 여기서는 **start**를 설정하여 서비스를 시작합니다.

15-2.Vagrant로 CoreOS설치하기

`$public_ipv4`, `$private_ipv4`

`$public_ipv4`, `$private_ipv4`는 Vagrant 뿐만 아니라 Amazon EC2, Google Compute Engine, OpenStack, Rackspace, DigitalOcean에서도 사용할 수 있고, 자동으로 현재 서버의 공인 IP 주소와 사설 IP 주소를 설정해줍니다.

메타데이터 설정하기

메타데이터를 설정하려면 다음과 같이 `fleet`에 `metadata` 항목을 추가하면 됩니다. `fleetctl`을 사용하여 메타데이터가 일치하는 노드에 유닛이 실행되도록 할 수 있습니다.

```
coreos:
  fleet:
    public-ip: $public_ipv4
    metadata: region=ap-northeast-1,disk=ssd,platform=cloud,provider=amazon
```

메타데이터는 특별한 규칙은 없으며 각자 편한대로 설정하면 됩니다. 각 메타데이터는 ,(콤마)로 구분합니다.

15-2.Vagrant로 CoreOS설치하기

config.rb

```
$num_instances=3  
$update_channel='stable'
```

coreos-vagrant 디렉터리 아래에 **config.rb** 파일로 저장합니다.

\$num_instances에 3을 설정하여 가상 머신을 3개 생성합니다.

\$update_channel에 **stable**을 설정하여 **Stable** 버전을 사용합니다.

user-data파일의 **discovery URL**을
config.rb에서 자동으로 설정
가능한 코드가 있습니다.

15-2.Vagrant로 CoreOS설치하기

이제 **coreos-vagrant** 디렉터리에서 다음 명령을 실행하여 가상 머신을 생성합니다

```
$ sudo vagrant up
```

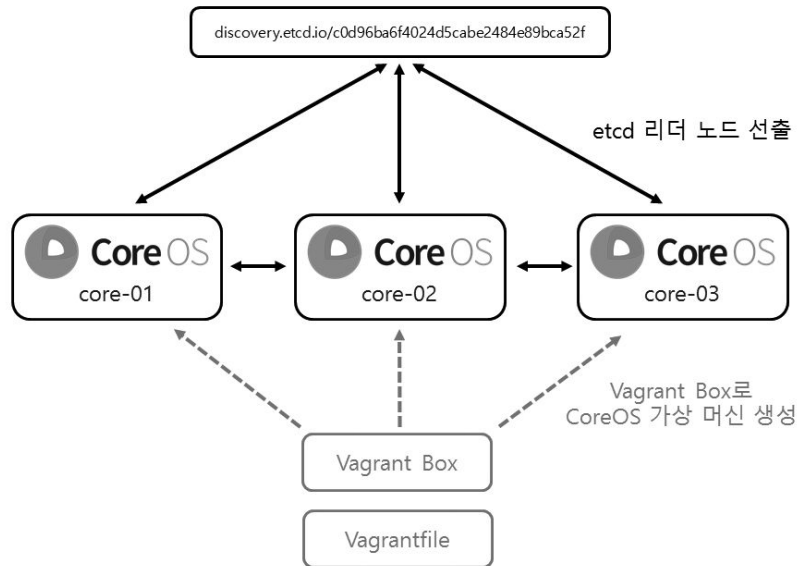
잠시 기다리면 가상 머신이 생성됩니다

이제 **vagrant status** 명령을 입력합니다.

```
$ sudo vagrant status
Current machine states:
```

```
core-01      running (virtualbox)
core-02      running (virtualbox)
core-03      running (virtualbox)
```

```
This environment represents multiple VMs. The VMs are all listed
above with their current state. For more information about a specific
VM, run `vagrant status NAME`.
```



15-2.Vagrant로 CoreOS설치하기

CoreOS가 설치된 core-01, core-02, core-03 가상 머신이 표시됩니다.

`vagrant ssh` 명령을 사용하여 core-01 가상 머신에 SSH로 접속해봅니다.

```
$ sudo vagrant ssh core-01
Last login: Sun Sep  7 11:40:28 2014 from 10.0.2.2
CoreOS (stable)
core@core-01 ~ $
```

`vagrant ssh <가상 머신 이름>` 형식이며, 간단하게 core-01 가상 머신에 접속했습니다.

15-3.etcd 사용하기

vagrant를 이용하여 **CoreOS** 가상 머신 3개로 클러스터를 구성해보았습니다.
이제 이 클러스터에서 **etcd**를 사용하여 데이터를 공유하는 방법을 알아보겠습니다.

터미널(Windows의 명령 프롬프트, PowerShell, Mac OS X의 터미널)을 2개
실행하고 **coreos-vagrant** 디렉터리로 이동한 뒤 **vagrant ssh** 명령으로
CoreOS 가상 머신에 접속합니다.

첫 번째 터미널

```
~$ cd coreos-vagrant  
~/coreos-vagrant$ vagrant ssh core-01
```

두 번째 터미널

```
~$ cd coreos-vagrant  
~/coreos-vagrant$ vagrant ssh core-02
```


15-3-1.etcd 키, 디렉터리 생성하기

core-01에서 다음 명령을 실행하여 키를 생성하고 값을 설정합니다.

core-01

```
$ etcdctl mk /hello world  
world
```

`etcdctl mk <etcd 키 경로> <값>` 형식이며, etcd 경로는 /로 시작합니다.

이제 core-02에서 다음 명령을 실행하여 키의 값을 가져옵니다.

core-02

```
$ etcdctl get /hello  
world
```

`etcdctl get <etcd 키 경로>` 형식입니다. 앞에서 설정한 값 **world**가 출력됩니다.

마찬가지로 core-01에서 디렉터를 생성하면 core-02에서도 볼 수 있습니다.

core-01

```
$ etcdctl mkdir /hello-dir
```

`etcdctl mk <etcd 디렉터리 경로>` 형식입니다.

15-3-2.etcd키, 디렉터리 목록 출력하기

etcd 키, 디렉터리 목록 출력하기

etcd는 일반 파일 시스템처럼 디렉터리를 가지고 있습니다. 다음 명령을 실행하여 키와 디렉터리를 출력합니다.

core-01

```
$ etcdctl ls / --recursive
/coreos.com
/coreos.com/updateengine
/coreos.com/updateengine/rebootlock
/coreos.com/updateengine/rebootlock/semaphore
/hello
```

`etcdctl ls <etcd 경로>` 형식이며 `--recursive` 옵션을 사용하면 하위 디렉터리와 키를 모두 출력합니다. 여기서 `/coreos.com` 디렉터리는 CoreUpdate(유료 서비스) 관련 설정입니다.

etcd 설정 값은 `_etcd` 디렉터리에 들어있고 `etcdctl ls` 명령으로는 보이지 않습니다. 다음 명령을 실행하여 `_etcd` 디렉터리의 내용을 살펴봅니다.

core-01

```
$ etcdctl ls /_etcd --recursive
/_etcd/machines
/_etcd/machines/b3318c7b2f63466e9f4065eacbda2b18
/_etcd/machines/d80aaff50718476ab5057c00804ee59b
/_etcd/machines/6960c4ce6f6b44518687a32d5e39dec9
/_etcd/config
```

CoreOS 가상 머신을 3개 생성했으므로 `/_etcd/machines` 디렉터리 아래에 키가 3개 들어있습니다.

15-3-3.etcd 키, 디렉터리 자동 삭제 설정하기

etcd 키, 디렉터리 자동 삭제 설정하기

etcd의 키와 디렉터리는 일정 시간이 지나면 자동으로 삭제되도록 설정할 수 있습니다. `etcdctl mk` 명령으로 키를 생성한 뒤 `etcdctl ls` 명령으로 /의 목록을 출력합니다.

core-01

```
$ etcdctl mk /hello2 world --ttl 20
world
$ etcdctl ls /
/coreos.com
/hello2
/hello-dir
```

`etcdctl mk <etcd 경로> <값> --ttl <초>` 형식입니다. 키를 생성하고 바로 /의 목록을 출력해보면 `/hello2` 키가 표시됩니다.

20초가 지난 뒤 다시 /의 목록을 출력합니다.

core-01

```
$ etcdctl ls /
/coreos.com
```

자동 삭제 설정을 했기 때문에 `/hello2` 키가 삭제되었습니다. 디렉터리도 마찬가지로 `etcdctl mkdir` 명령에 `--ttl` 옵션만 붙여주면 됩니다.

core-01

```
$ etcdctl mkdir /hello-dir2 --ttl 20
```

15-3-4.etcd키 감시하기

`etcdctl watch` 명령을 사용하면 키가 추가/삭제되거나 값이 변경되는 시점을 바로 알 수 있습니다.

core-01에서 키를 생성하고 `etcdctl watch` 명령을 실행합니다.

core-01

```
$ etcdctl mk /hello3 world
world
$ etcdctl watch /hello3
```

`etcdctl watch <etcd 경로>` 형식입니다.

이제 core-02에서 `/hello3` 키의 값을 `abcd1234`로 변경합니다.

core-02

```
$ etcdctl set /hello3 abcd1234
abcd1234
```

`etcdctl set <etcd 경로> <값>` 형식입니다.

core-01을 보면 다음과 같이 `etcdctl watch` 명령이 종료되고 `abcd1234`가 출력됩니다.

core-01

```
$ etcdctl watch /hello3
abcd1234
```

`etcdctl exec-watch` 명령을 사용하면 키가 변경되었을 때 특정 명령을 실행할 수 있습니다.

core-01

```
$ etcdctl exec-watch /hello3 -- sh -c "echo hello3 modified"
hello3 modified
```

`etcdctl exec-watch <etcd 경로> -- sh -c "<명령>"` 형식입니다. 여기서 `/hello3` 키를 변경하면 `sh`를 통해 `echo hello3 modified`가 실행됩니다.

15-3-5.etcd 기타 명령

etcd 기타 명령

키와 디렉터리를 삭제하는 명령은 다음과 같습니다.

```
$ etcdctl rm /hello3  
$ etcdctl rmdir /hello-dir
```

`etcdctl rm <etcd 키 경로>`, `etcdctl rmdir <etcd 디렉터리 경로>` 형식입니다.

키를 디렉터리로 만드는 명령은 다음과 같습니다.

```
$ etcdctl setdir /hello3
```

`etcdctl setdir <etcd 키 경로>` 형식입니다.

이미 생성되어 있는 키의 값이나 설정을 변경하거나 디렉터리의 설정을 변경하는 명령은 다음과 같습니다.

```
$ etcdctl update /hello abcd1234 --ttl 20  
$ etcdctl updatedir /hello-dir --ttl 20
```

- `etcdctl update <etcd 키 경로> <값> --ttl <초>` 형식입니다. `--ttl` 옵션을 생략하여 값만 변경할 수 있습니다.
- `etcdctl updatedir <etcd 디렉터리 경로> --ttl <초>` 형식입니다. `--ttl` 옵션은 생략할 수 없습니다.

15-4.fleet 사용하기

fleet 사용하기

이번에는 fleet을 사용하여 클러스터에 서비스를 실행해보겠습니다. 터미널(Windows의 명령 프롬프트, PowerShell, Mac OS X의 터미널)을 3개 실행하고 **coreos-vagrant** 디렉터리로 이동한 뒤 **vagrant ssh** 명령으로 CoreOS 가상 머신에 접속합니다.

첫 번째 터미널

```
~$ cd coreos-vagrant  
~/coreos-vagrant$ vagrant ssh core-01
```

두 번째 터미널

```
~$ cd coreos-vagrant  
~/coreos-vagrant$ vagrant ssh core-02
```

세 번째 터미널

```
~$ cd coreos-vagrant  
~/coreos-vagrant$ vagrant ssh core-03
```

15-4-1.fleet 머신 목록 출력하기

다음 명령을 실행하여 클러스터에 속한 머신(서버, 노드) 목록을 출력합니다.

```
$ fleetctl list-machines
MACHINE      IP          METADATA
6960c4ce...  172.17.8.103 -
b3318c7b...  172.17.8.101 -
d80aaff5...  172.17.8.102 -
```

머신 ID와 IP 주소, 메타데이터가 출력됩니다. `--full` 옵션을 사용하면 머신 ID를 모두 출력합니다.

15-4-2.fleet으로 유닛 실행하기

core-01에서 다음 내용을 `hello.service` 파일로 저장합니다.

`fleet`로 실행하는 유닛 파일(`.service`)은 `/etc/systemd/system` 디렉터리가 아닌 다른 디렉터리에 있어도 상관없습니다.

~/hello.service

```
[Unit]
Description=Hello Service
Requires=docker.service
After=docker.service

[Service]
ExecStartPre=/usr/bin/docker kill hello
ExecStartPre=/usr/bin/docker rm hello
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
```

- Unit: 유닛 실행 설정입니다.
 - Requires, After: Docker를 사용할 것이므로 **docker.service**를 설정합니다.
- Service: 각 상황에 따라 실행할 명령을 설정합니다.
 - ExecStartPre: 메인 명령 실행 전의 준비 작업입니다. `docker kill hello`, `docker rm hello` 명령을 실행하여 **hello** 컨테이너가 실행되고 있다면 종료 후 삭제합니다. `ExecStartPre=-`처럼 `=뒤에 -`를 붙여주면 에러가 발생하더라도 그냥 넘어갑니다. 여기서 **hello** 컨테이너가 없을 때 컨테이너를 종료하거나 삭제하려고 하면 에러가 발생하기 때문에 반드시 `-`를 붙여줍니다.
 - ExecStart: 유닛이 시작될 때 실행할 메인 명령입니다. `busybox` 이미지로 **hello** 컨테이너를 생성하고 1초마다 **Hello World**를 출력합니다.
 - ExecStop: 유닛이 종료될 때 실행할 명령입니다. `docker stop hello` 명령을 실행하여 컨테이너를 정지합니다.

15-4-2.fleet으로 유닛 실행하기

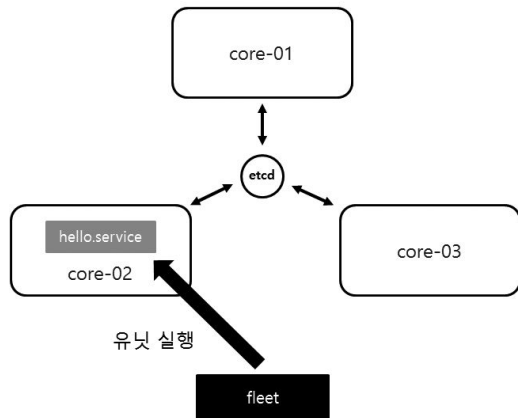
hello.service 파일이 있는 디렉터리에서 다음 명령을 실행합니다.

core-01

```
$ fleetctl start hello.service
Job hello.service launched on d80aaff5.../172.17.8.102
```

`fleetctl start <유닛 파일>` 형식입니다. 클러스터의 core-01, core-02, core-03 중에서 아무곳에 유닛이 실행됩니다. 저는 core-02(d80aaff5, 172.17.8.102)에 hello.service 유닛이 실행되었습니다.

`fleetctl start` 명령으로 유닛을 실행하면 `hello.service` 파일의 내용이 클러스터의 모든 노드에 공유됩니다.



기타 유닛 옵션

유닛 파일에서 `docker pull` 명령으로 이미지를 받을 때 시간이 오래 걸리면 유닛 실행이 실패하게 됩니다. 다음과 같이 `TimeoutStartSec`에 0을 설정하면 명령 실행 시간이 오래 걸리더라도 유닛 생성에 실패하지 않습니다.

```
[Service]
TimeoutStartSec=0
```

다음은 유닛이 비정상적으로 종료되었을 때 자동으로 재시작하는 옵션입니다. `Restart`에 `always`를 설정하면 계속 재시작합니다. `RestartSec`에 시간을 설정하면 재시작하기 전에 특정 시간 동안 대기합니다. 예) 5min 20s, 100ms, 10s

```
[Service]
Restart=always
RestartSec=5s
```

15-4-3.fleet유닛 목록 출력하기

다음 명령을 실행하여 클러스터상의 유닛 목록을 출력합니다.

core-01

```
$ fleetctl list-units
```

UNIT	DSTATE	TMACHINE	STATE	MACHINE	ACTIVE
hello.service	launched	d80aaff5.../172.17.8.102	launched	d80aaff5.../172.17.8.102	active

앞에서 **hello.service** 유닛을 실행했기 때문에 core-02(d80aaff5, 172.17.8.102)에 **hello.service** 유닛이 실행된 모습을 볼 수 있습니다. 각자 상황에 따라 유닛이 실행된 노드는 달라질 수 있습니다.

15-4-4.fleet 유닛 상태 확인하기

hello.service 유닛이 실행된 노드에서 다음 명령을 실행합니다. 저는 hello.service 유닛이 core-02에 실행되었기 때문에 core-02에서 명령을 실행하겠습니다.

core-02

```
$ fleetctl status hello.service
• hello.service - Hello Service
  Loaded: loaded (/run/fleet/units/hello.service; linked-runtime)
  Active: active (running) since Mon 2014-09-08 13:30:42 UTC; 2min 43s ago
  Process: 1115 ExecStartPre=/usr/bin/docker rm hello (code=exited, status=1/FAILURE)
  Process: 1059 ExecStartPre=/usr/bin/docker kill hello (code=exited, status=1/FAILURE)
  Main PID: 1129 (docker)
  CGroup: /system.slice/hello.service
          └─1129 /usr/bin/docker run --name hello busybox /bin/sh -c while true; do echo Hello World; sleep 1; d
one

Sep 08 13:33:16 core-02 docker[1129]: Hello World
Sep 08 13:33:17 core-02 docker[1129]: Hello World
Sep 08 13:33:18 core-02 docker[1129]: Hello World
Sep 08 13:33:19 core-02 docker[1129]: Hello World
Sep 08 13:33:20 core-02 docker[1129]: Hello World
Sep 08 13:33:21 core-02 docker[1129]: Hello World
Sep 08 13:33:22 core-02 docker[1129]: Hello World
Sep 08 13:33:23 core-02 docker[1129]: Hello World
Sep 08 13:33:24 core-02 docker[1129]: Hello World
Sep 08 13:33:25 core-02 docker[1129]: Hello World
```

`fleetctl status <유닛 이름>` 형식입니다. 현재 유닛의 상황과 로그가 표시됩니다.

`fleetctl journal` 명령을 사용하면 유닛의 로그만 출력할 수 있습니다.

core-02

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

`fleetctl journal <유닛 이름>` 형식입니다. `-f` 옵션을 사용하면 로그를 실시간으로 출력합니다.

15-4-5.fleet 자동 복구 확인하기

fleet는 노드에 장애가 발생하면 해당 노드에서 실행되던 유닛을 다른 노드에서 실행해줍니다. 이번에는 **hello.service** 유닛이 실행되는 노드(저는 **core-02**)에 장애가 발생했다고 가정해보고, 해당노드에서 다음 명령을 실행하여 가상 머신을 종료합니다.

core-02

```
$ sudo halt
```

이제 다른 노드에서 서버(머신) 목록을 출력합니다.

core-01

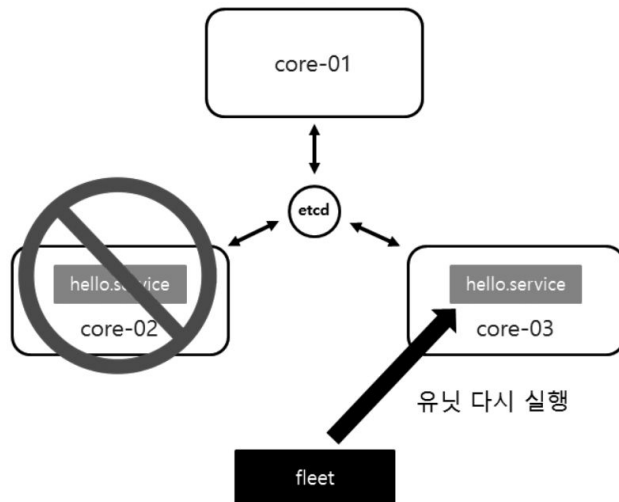
```
$ fleetctl list-machines
MACHINE      IP            METADATA
6960c4ce...  172.17.8.103 -
b3318c7b...  172.17.8.101 -
```

core-02를 종료했기 때문에
core-01(b3318c7b, 172.17.8.101),
core-03(6960c4ce, 172.17.8.103)만 표시됩니다.

core-01

```
$ fleetctl list-units
UNIT      DSTATE    TMACHINE                STATE    MACHINE                ACTIVE
hello.service  launched  6960c4ce.../172.17.8.103  launched  6960c4ce.../172.17.8.103  active
```

저는 **hello.service** 유닛이 core-03(6960c4ce, 172.17.8.103)에 다시 실행되었습니다. 각자 상황에 따라 유닛이 실행되는 노드는 달라질 수 있습니다. 이렇게 CoreOS는 fleet을 이용하여 고가용성(High Availability)를 제공합니다.



15-4-6.fleet전용 옵션 사용하기

지금까지 클러스터에서 유닛을 실행되는 노드가 무작위로 정해졌습니다.

이번에는 유닛이 실행되는 노드를 설정할 수 있는 **fleet** 전용 옵션에 대해 알아보겠습니다.

systemd 유닛 파일(.service)에서 **fleet** 전용 옵션은 **[X-Fleet]** 섹션을 사용합니다.

옵션 이름	설명
MachineID	특정 머신에서 유닛을 실행합니다. 머신 ID는 축약된 형태가 아닌 완전한 형태를 설정해야 합니다. 예) <code>MachineID=6960c4ce6f6b44518687a32d5e39dec9</code>
MachineOf	특정 유닛이 실행되고 있는 머신에서 유닛을 실행합니다. 예) <code>MachineOf=hello.service</code>
MachineMetadata	지정한 메타데이터와 일치하는 머신에서 유닛을 실행합니다. 예) <code>MachineMetadata=region=ap-northeast-1</code>
Conflicts	특정 유닛이 실행되고 있는 머신에서는 유닛을 실행하지 않습니다. 예) <code>Conflicts=hello.service</code> <code>hello.*.service</code> 처럼 여러 유닛을 설정할 수도 있습니다.
Global	클러스터의 모든 노드에 유닛을 실행합니다. 옵션 중에서 MachineMetadata만 사용할 수 있고 나머지는 무시됩니다. 예) <code>Global=true</code>

표 15-1 fleet 전용 옵션

15-4-6.fleet전용 옵션 사용하기

특정 머신에서 유닛을 실행하려면 다음과 같이 유닛 파일을 작성합니다.

완전한 형태의 머신 ID는 `fleetctl list-machines -full` 명령으로 얻을 수 있습니다.

~/hello.service

```
[Unit]
Description=Hello Service
Requires=docker.service
After=docker.service

[Service]
ExecStartPre=/usr/bin/docker kill hello
ExecStartPre=/usr/bin/docker rm hello
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

[X-Fleet]
MachineID=6960c4ce6f6b44518687a32d5e39dec9
```

15-4-6.fleet전용 옵션 사용하기

`fleetctl start` 명령으로 유닛을 실행하면 `core-03(6960c4ce, 172.17.8.103)`에 실행됩니다

다음은 특정 유닛(`hello.service`)이 실행되고 있는 머신에서 유닛을 실행하는 예제입니다.

이러한 방식은 웹 서버, DB 등의 IP 주소와 포트를 알아내는 사이드킥(`sidekick`) 모델에서 활용됩니다.

```
$ fleetctl start hello.service
Job hello.service launched on 6960c4ce.../172.17.8.103
```

world.service

```
[Unit]
Description=World Service
Requires=docker.service
After=docker.service

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

[X-Fleet]
MachineOf=hello.service
```

15-4-6.fleet전용 옵션 사용하기

다음은 지정한 메타데이터와 일치하는 머신에서 유닛을 실행하는 예제입니다.
리전이 **ap-northeast-1**이고 클라우드 플랫폼이 **amazon**인 머신에서 유닛을 실행합니다.

hello-nginx.service

```
[Unit]
Description=Hello Nginx Service
Requires=docker.service
After=docker.service

[Service]
ExecStartPre=/usr/bin/docker kill hello-nginx
ExecStartPre=/usr/bin/docker rm hello-nginx
ExecStart=/usr/bin/docker run --name hello-nginx -p 80:80 nginx:latest
ExecStop=/usr/bin/docker stop hello-nginx

[X-Fleet]
MachineMetadata="region=ap-northeast-1" "provider=amazon"
```

us-east-1과 us-west-1에 유닛을 실행하고 싶다면 다음과 같이 설정합니다.

```
[X-Fleet]
MachineMetadata=region=us-east-1
MachineMetadata=region=us-west-1
```


15-4-6.fleet전용 옵션 사용하기

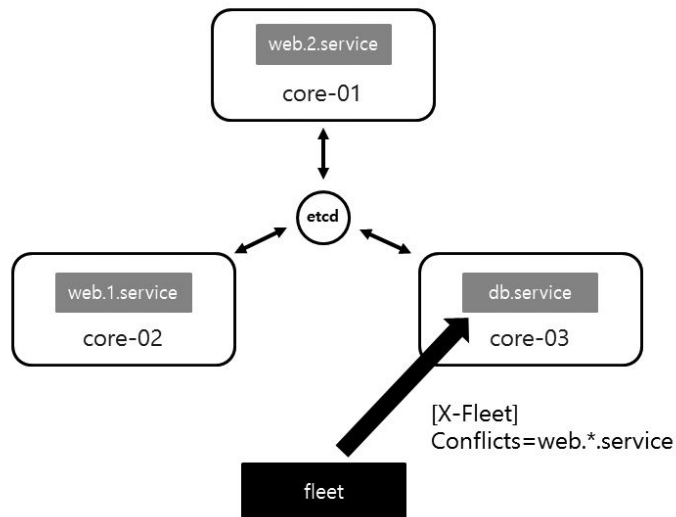
다음은 특정 유닛이 실행되고 있는 머신을 피해서 다른 머신에서 유닛을 실행하는 예제입니다.
웹 서버 유닛과 DB 유닛이 있을 때, DB 유닛은 웹 서버 유닛이 실행되지 않은 머신에서 실행합니다.

db.service

```
[Unit]
Description=DB Service
Requires=docker.service
After=docker.service

[Service]
ExecStartPre=/usr/bin/docker kill db
ExecStartPre=/usr/bin/docker rm db
ExecStart=/usr/bin/docker run --name db -p 27017:27017 mongo:latest
ExecStop=/usr/bin/docker stop db

[X-Fleet]
Conflicts=web.*.service
```



15-4-7.fleet 유닛 파일 템플릿 활용하기

systemd는 유닛 파일 하나로 여러 개의 인스턴스를 생성하는 템플릿 기능을 제공합니다. 여기에 **fleet**을 활용하면 유닛 파일 하나로 여러 노드에 유닛을 생성할 수 있습니다.

- 유닛 파일 템플릿의 파일명은 **<유닛 이름>@.<확장자>** 형식입니다. 예) **hello@.service**
- 유닛 파일 템플릿으로 유닛을 생성할 때는 **<유닛 이름>@<인스턴스 이름>.<확장자>** 형식으로 생성합니다. 예) **hello@world.service**, **hello@1.service**

~/hello@.service

```
[Unit]
Description=Hello Service
Requires=docker.service
After=docker.service

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

[x-Fleet]
Conflicts=hello@*.service
```

- %i: 인스턴스 이름입니다. @뒤에 오는 인스턴스 이름이 대입됩니다.
- Conflicts: hello@*.service로 지정하여 유닛이 겹치지 않고 각각 다른 노드에 실행되도록 합니다.

15-4-7.fleet 유닛 파일 템플릿 활용하기

다음 명령을 실행하여 hello@.service 파일을 클러스터에 저장합니다. 그리고 클러스터에 저장된 유닛 파일 목록을 출력합니다.

core-01

```
$ fleetctl submit hello@.service
$ fleetctl list-unit-files
UNIT          HASH      DSTATE
hello@.service 1aeee55 inactive
```

- `fleetctl submit <유닛 파일>` 형식입니다. 유닛을 실행하지는 않고 유닛 파일의 내용을 클러스터에 저장만 합니다.
- `fleetctl list-unit-files` 명령은 클러스터에 저장된 유닛 파일 목록을 출력합니다. 클러스터에 저장된 hello@.service 파일이 표시됩니다.

15-4-7.fleet 유닛 파일 템플릿 활용하기

이제 다음 명령을 실행하여 유닛을 실행합니다.

core-01

```
$ fleetctl start hello@{1..3}.service
Unit hello@1.service launched on d80aaff5.../172.17.8.102
Unit hello@2.service launched on 6960c4ce.../172.17.8.103
Unit hello@3.service launched on b3318c7b.../172.17.8.101
```

`fleetctl start <유닛 이름>@{시작 숫자...끝 숫자}.service` 형식입니다.

`{1..3}`과 같이 입력하면 1, 2, 3 인스턴스 3개를 생성합니다.

`fleetctl start hello@1.service`처럼 유닛을 개별적으로 생성할 수도 있습니다.

명령을 실행하면 유닛이 특정 노드에 몰리지 않고 `core-01`, `core-02`, `core-03`에 각각 생성되었습니다.

15-4-8.fleet 사이드킥 모델 활용하기

CoreOS 클러스터에서 **fleet**을 사용하면 머신 ID, 메타데이터, 특정 유닛의 실행 유무에 따라 유닛을 실행할 노드를 선택하게 됩니다. 즉 각 노드의 IP 주소로 유닛을 실행한 것이 아니기 때문에 유닛의 IP 주소는 알 수가 없습니다. 이때 IP 주소를 얻어오는 방식이 사이드킥(sidekick) 모델입니다.

앞에서 **etcd**를 이용하여 노드끼리 데이터를 공유했습니다.
사이드킥 모델은 **etcd**를 이용하여 IP 주소를 공유합니다.
core-01에서 다음 내용을 **web.service** 파일로 저장합니다.

~/web.service

```
[Unit]
Description=Web Service
Requires=docker.service
After=docker.service

[Service]
ExecStartPre=/usr/bin/docker kill web
ExecStartPre=/usr/bin/docker rm web
ExecStart=/usr/bin/docker run --name web -p 80:80 nginx:latest
ExecStop=/usr/bin/docker stop web
```

15-4-8.fleet 사이드kick 모델 활용하기

다음 내용을 web-discovery.service 파일로 저장합니다.

~/web-discovery.service

```
[Unit]
Description=Announce Web
BindsTo=web.service

[Service]
EnvironmentFile=/etc/environment
ExecStart=/bin/sh -c \
    "while true; \
    do \
        etcdctl set /services/web/nginx \
        '{ \"host\": \"${COREOS_PUBLIC_IPV4}\", \"port\": 80 }' \
        --ttl 60; \
        sleep 45; \
    done"
ExecStop=/usr/bin/etcdctl rm /services/web/nginx

[X-Fleet]
MachineOf=web.service
```

- BindsTo: **web.service**를 지정하여 **web.service** 유닛이 종료되면 현재 유닛도 함께 종료합니다.
- EnvironmentFile: 현재 유닛의 환경 변수를 설정할 파일입니다. **/etc/environment** 파일에 현재 노드의 공인 IP 주소(**COREOS_PUBLIC_IPV4**)와 사설 IP 주소(**COREOS_PRIVATE_IPV4**)가 저장되어있으므로 반드시 설정합니다.
- ExecStart: etcd에 IP 주소와 포트 번호를 등록하는 셸 스크립트를 실행합니다.
 - **etcdctl set** 명령으로 **/services/web/nginx**에 키를 생성하고 IP 주소와 포트 번호를 저장합니다. IP 주소는 **COREOS_PUBLIC_IPV4** 변수에 저장되어 있습니다.
 - **--ttl** 옵션을 사용하여 60초 뒤에 키가 삭제되도록 설정합니다.
 - **sleep 45**로 45초마다 **etcdctl set** 명령을 반복 실행하여 키가 삭제되기 전에 내용을 갱신합니다. 정상적인 노드는 키를 계속 유지할 수 있으며 중단된 노드는 **etcdctl set** 명령을 실행할 수 없어서 키가 삭제되므로 노드 상태를 확실하게 알 수 있습니다. 또한, 이렇게 반복 실행하면 현재 노드가 중단된 뒤 다른 노드에서 유닛이 다시 실행될 때 **etcd**에 새 IP 주소를 갱신할 수 있습니다.
- ExecStop: 유닛이 정지되면 **etcd**의 **/services/web/nginx** 키를 삭제합니다.
- MachineOf: **web.service**를 지정하여 **web.service** 유닛이 실행되고 있는 노드에서 현재 유닛을 실행합니다.

15-4-8.fleet 사이드킥 모델 활용하기

`fleetctl start` 명령으로 **web.service**, **web-discovery.service** 유닛을 실행합니다.

core-01

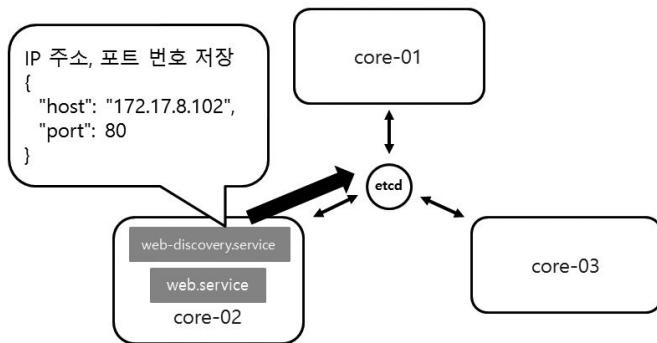
```
$ fleetctl start web.service
Job web.service launched on d80aaff5.../172.17.8.102
$ fleetctl start web-discovery.service
Job web-discovery.service launched on d80aaff5.../172.17.8.102
```

web.service, **web-discovery.service** 유닛이 core-02(d80aaff5, 172.17.8.102)에 실행되었습니다.

이제 **etcdctl get** 명령을 사용하여 IP 주소와 포트 번호를 출력합니다.

core-01

```
$ etcdctl get /services/web/nginx
{ "host": "172.17.8.102", "port": 80 }
```



15-4-9.fleet 기타 명령

클러스터에서 유닛 파일을 삭제하는 명령은 다음과 같습니다.

```
$ fleetctl destroy hello.service
```

`fleetctl destroy <유닛 이름>` 형식입니다. 클러스터에서 유닛 파일을 삭제하면 모든 노드에서 유닛 파일이 삭제됩니다.

클러스터에 저장된 유닛 파일의 내용을 출력하는 명령은 다음과 같습니다.

```
$ fleetctl cat hello.service
```

`fleetctl cat <유닛 이름>` 형식입니다. 로컬에 저장된 유닛 파일이 아닌 클러스터에 저장된 유닛 파일의 내용을 출력합니다.

유닛을 노드에 할당만 하고 실행은 하지 않는 명령은 다음과 같습니다.

```
$ fleetctl load hello.service
$ fleetctl list-units
UNIT          DSTATE  TMACHINE          STATE  MACHINE          ACTIVE
hello.service loaded  6960c4ce.../172.17.8.103 loaded  6960c4ce.../172.17.8.103 inactive
```

`fleetctl load <유닛 이름>` 형식입니다. `fleetctl list-units` 명령으로 유닛 목록을 출력해보면 노드에 할당만 되었고, 실행이 되지 않은 inactive 상태입니다.

유닛을 노드에서 삭제하는 명령은 다음과 같습니다.

```
$ fleetctl unload hello.service
$ fleetctl list-units
UNIT          DSTATE  TMACHINE  STATE  MACHINE  ACTIVE
hello.service inactive -      inactive -      -
```

`fleetctl unload <유닛 이름>` 형식입니다. `fleetctl list-units` 명령으로 유닛 목록을 출력해보면 할당된 노드가 없습니다. `fleetctl destroy` 명령과는 달리 클러스터에서 유닛 파일을 삭제하지 않습니다.