Hey everyone,

TL;DR

I hacked something together in order to create a Kubernetes cluster on CoreOS (or Container Linux) using Vagrant and Ansible.

If you keep reading, I'm going to talk to you about Kubernetes, etcd, CoreOS, flannel, Calico, Infrastructure as Code and Ansible testing strategies. It's gonna be super fun.

If you want to try it:

```
git clone https://github.com/sebiwi/kubernetes-coreos
cd kubernetes-coreos
make up
```

This will spin up 4 VMs: an etcd node, a Kubernetes Master node, and two Kubernetes Worker nodes. You can modify the size of the cluster by hacking on the Vagrantfile and the Ansible inventory.

You will need **Ansible** 2.2, **Vagrant**, **Virtualbox** and **kubectl**. You will also need **molecule** and **docker-py**, if you want to run the tests.

Why?

The last time I worked with Kubernetes was last year. Things have changed since then. A guy I know once said that in order to understand how something complex works, you need to build it up from scratch. I guess that's one of the main points of this project, really. S*cientia potentia est.* I also wanted to be able to test different Kubernetes features in a set up reminiscent of a production cluster. Minikube is great and all, but you don't actually get to see communication between different containers on different hosts, node failover scenarios, scheduling policies, or hardcore scale up procedures (with many nodes).

Finally, I thought it would be nice to explain how every Kubernetes component fits together, so everyone can understand what's under the hood. I keep getting questions like "what is *a Kubernetes"* at work, and if it is "*better than a Docker"*. You won't find the answer to the last question in this article, but at least you will (hopefully) understand how Kubernetes works. You can make up your own mind afterwards.

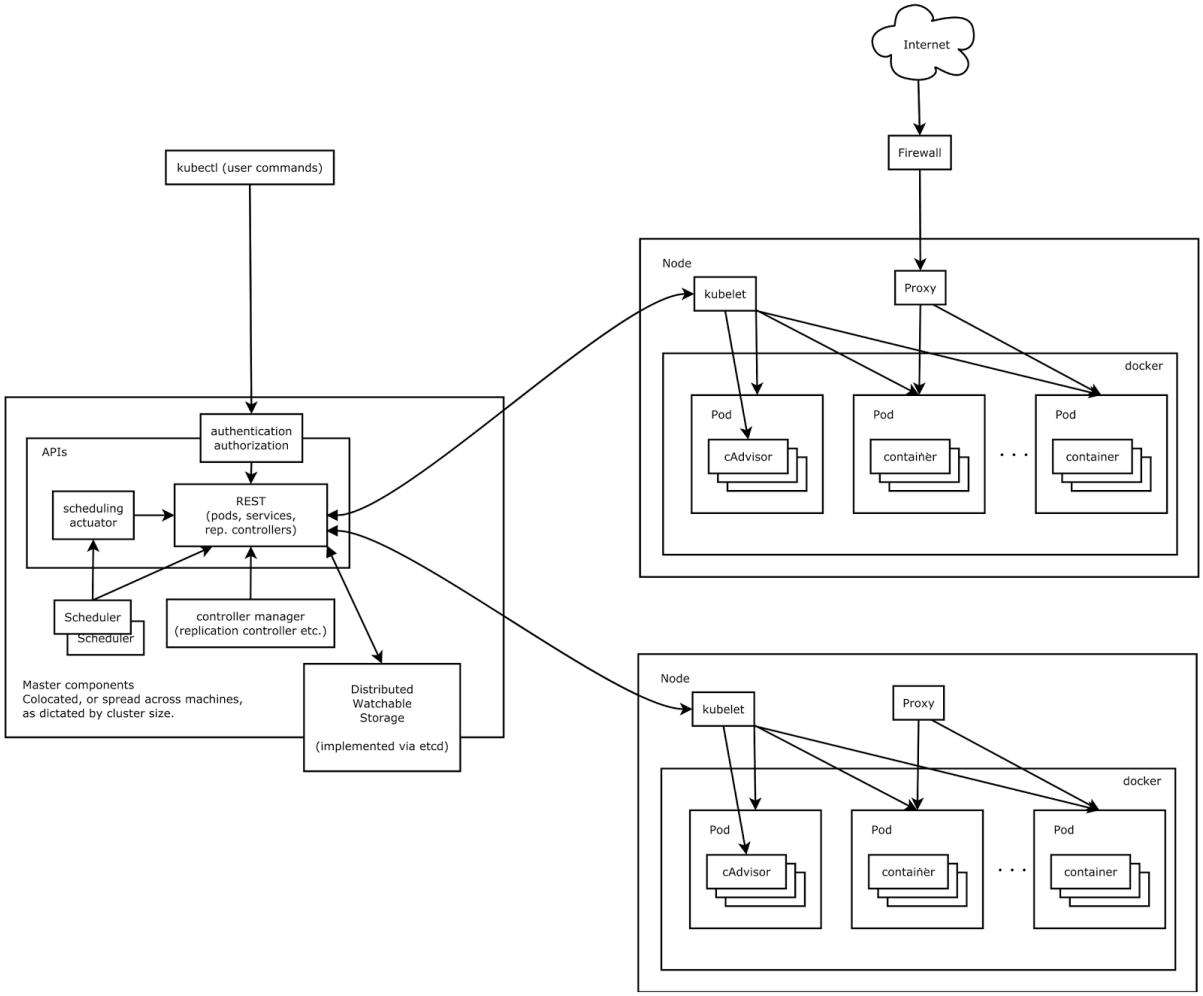
Okay, I was just passing by, but what is Kubernetes?

Kubernetes is a container cluster management tool. I will take a (not so) wild guess and assume that you've already heard about Docker and containers in general.
The thing is that Docker by itself will probably not suffice when using containers in production. What if your application is composed of multiple containers? You will need to be able to handle not only the creation of these containers, but also the communication between them. What if you feel that putting all your containers on the same host sucks, since if that host goes down, all your containers die with it? You will need to be able to deploy containers on many hosts, and also handle the communication between them, which translates into port mapping hell unless you're using an SDN solution. What about deploying a new version of your application without service interruption? What about container failure management, are you going to do go check on every container independently to see if it is healthy, and relaunch it manually if it is not? Grrrrrrrraaaaah.

Kubernetes is a tool you can use if you do not want to develop something specific in order to handle all the aforementioned issues. It can help you **pilot** your container cluster, hence its name, which means pilot or helmsman in greek.

Just a little heads up before we start talking about architecture: I will often talk about pods during these series. A pod is a group of one or more containers, that run in a shared context. A pod models an application-specific “logical host”, and theoretically it contains one or more applications that are tightly coupled: the kind of applications that you would have executed on the same physical or virtual host before containers. In practice, and for our use-case, you can just think of a pod as a container, since we will only have one container inside each pod.

A standard Kubernetes installation consists of both Master and Worker nodes:

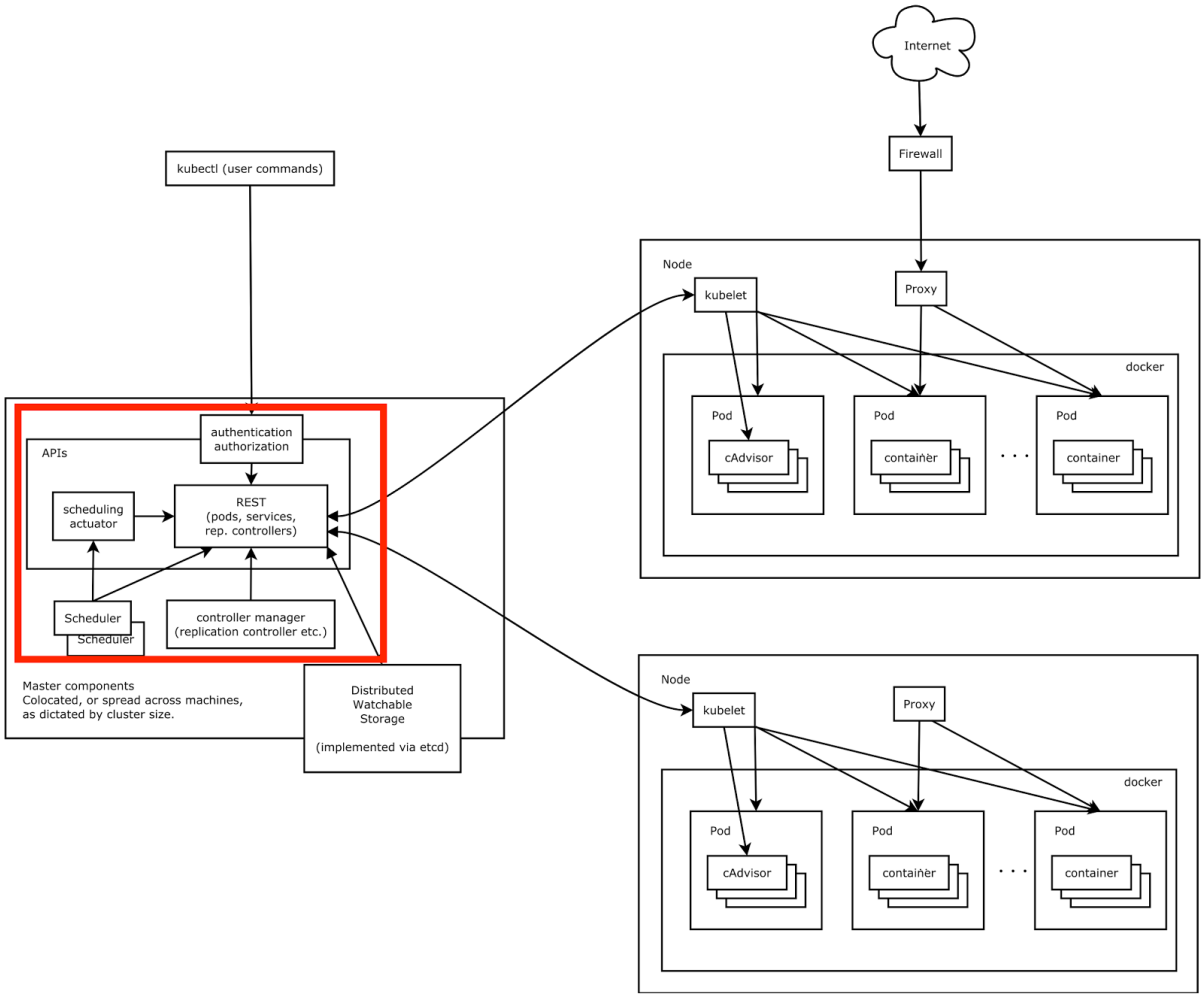


General Kubernetes Architecture, for real [1]

Wait, where is the Master node?

This architecture diagram specifies a set of master components, but not a Master node per-se. You will probably notice that the Distributed Watchable Storage is considered among these components. Nevertheless, we will not install it on the same node. Our Distributed Watchable Storage will be a separate component (more on this later).

So the Master node actually becomes everything that is inside the red square:



Here, see? [2]

All these components are part of the Kubernetes control pane. Keep in mind that you can have these on one single node, but you can also put them on many different ones. In our case we're putting them all together. So basically, we have an API server, a Scheduler, and a Controller Manager Server.

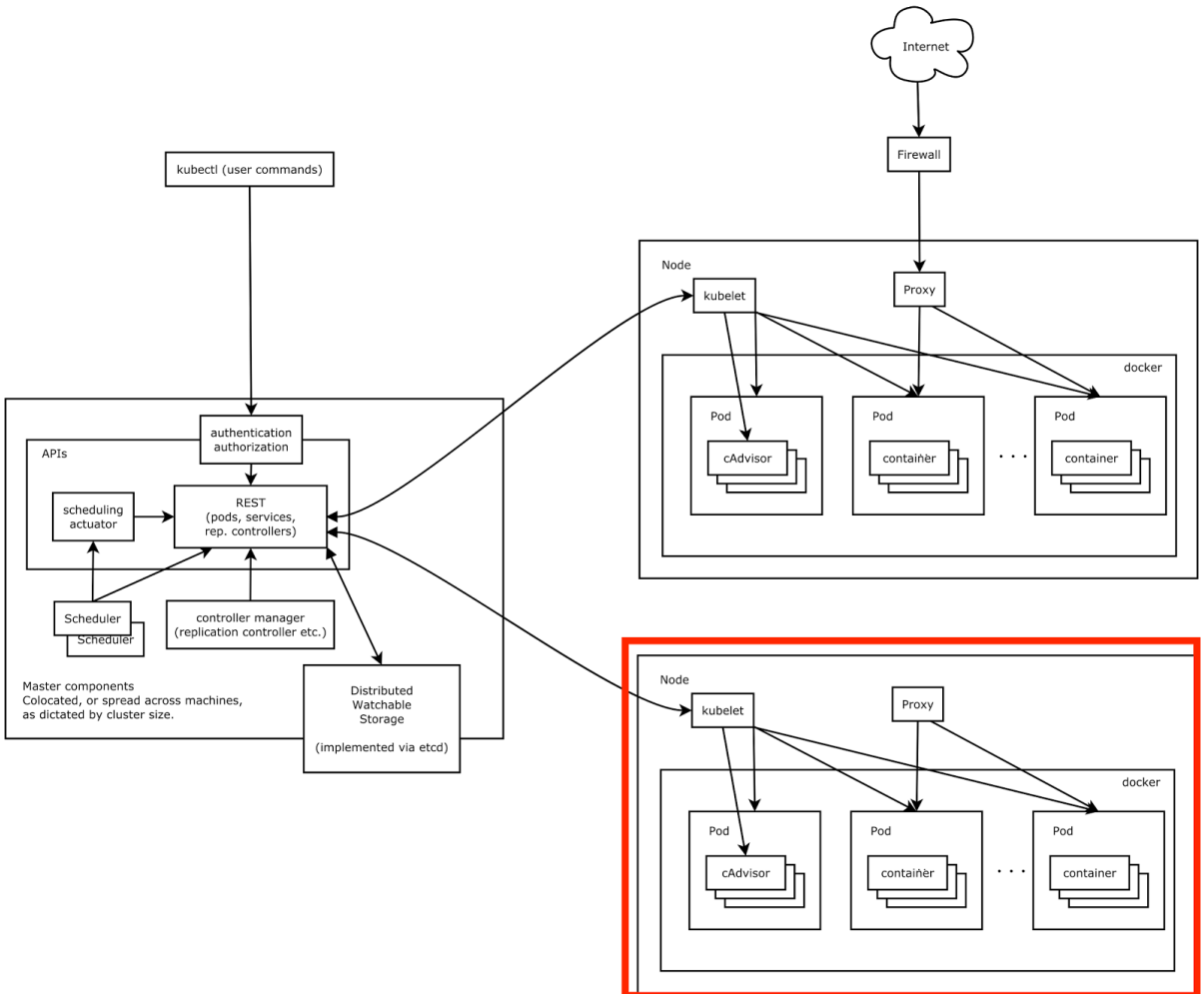
The API server exposes the Kubernetes API (duh). It processes REST operations, and then updates etcd consequently. The scheduler binds the unscheduled pods to a suitable worker node. If none are available, the pod remains unscheduled until a fitting node is found. The controller manager server does all the other cluster-level functions, such as endpoint creation, node discovery, and replication control. Many controllers are embedded into this controller manager, such as the endpoint controller, the node controller and the replication controller. It watches the shared state of the Kubernetes cluster using the API server, and makes changes on it with the intention of making the current state and the desired state of the cluster match.

What about the Worker node?

The Master node does not run any containers, it just handles and manages the cluster*. The nodes that actually run the containers are the Worker nodes.

*: This is not actually true in our case, but we will talk about that later.

The Worker node is composed of a kubelet, and a proxy (kube-proxy). You can see these components inside the red square, in the diagram below.



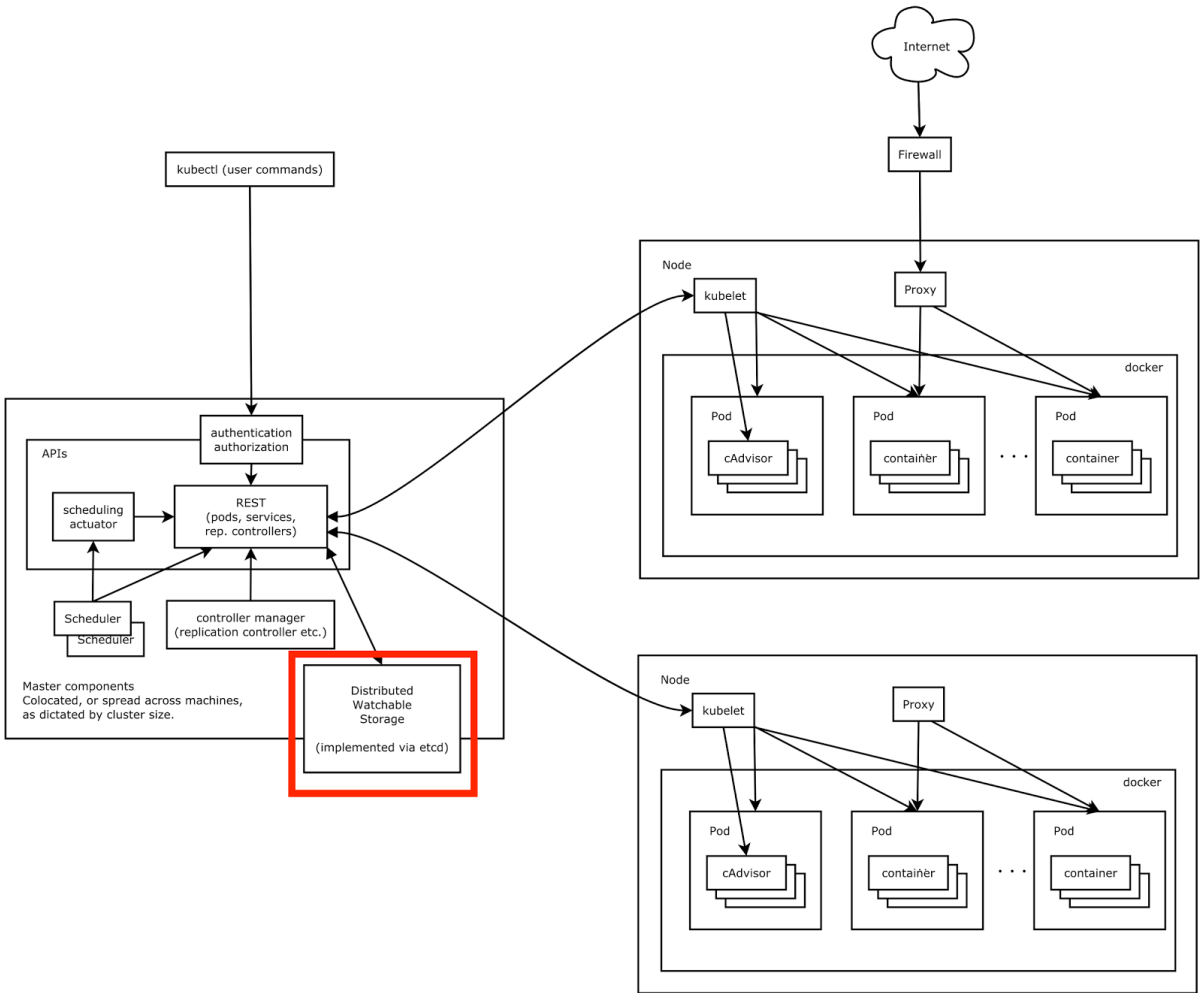
Right here [3]

The kubelet is the agent on the worker node that actually starts and stops the pods, and communicates with the Docker engine at a host level. This means it also manages the containers, the images and the associated volumes. It communicates with the API server on the Master node.

The kube-proxy redirects traffic directed to specific services and pods to their destination. It communicates with the API server too.

And what about that Distributed Watchable Storage thing?

Oh, you mean etcd. This is one of the components that is actually included in Container Linux, and developed by CoreOS. It is a distributed, fault tolerant key-value store used for shared configuration and service discovery. It actually means "something like etc, distributed on many hosts" . This sucks as a name because it is not a filesystem, but a key-value store. They are aware of it though. Just in case you missed it in the previous diagrams, it is the square inside the red square in the following diagram:



You never know [4]

All the persistent master state is stocked in etcd. Since components can actually "watch" components, they are able to realise that something has changed rather quickly, and then do something about it.

And that's what I deployed on CoreOS using Ansible. All of these things, and then some more.

That's rad, how did you do it?

You need to understand a little bit about Kubernetes networking before we get to that.

Kubernetes networking

As I said in the previous article, communication between pods that are hosted on different machines can be a little bit tricky. Docker will create by default a virtual bridge called "**docker0"** on the host machine, and it will assign a private network range to it.

```
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:b4:78:1f:41 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:b4ff:fe78:1f41/64 scope link
       valid_lft forever preferred_lft forever
```

Super bridge (172.17.0.1/16 in this case)

For each container that is created, a virtual ethernet device is attached to this bridge, which is then mapped to eth0 inside the container, with an ip within the aforementioned network range. Note that this will happen for each host that is running Docker, without any coordination between the hosts. Therefore, the network ranges might collide.
Because of this, containers will only be able to communicate with containers that are connected to the same virtual bridge. In order to communicate with other containers on other hosts, they must rely on port-mapping. This means that you need to assign a port on the host machine to each container, and then somehow forward all traffic on that port to that container. What if your application needs to advertise its own IP address to a container that is hosted on another node? It doesn't actually knows its real IP, since his local IP is getting translated into another IP and a port on the host machine. You can automate the port-mapping, but things start to get kinda complex when following this model.

That's why Kubernetes chose simplicity and skipped the dynamic port-allocation deal. It just assumes that all containers can communicate with each other without Network Address Translation (NAT), that all containers can communicate with each node (and vice-versa), and that the IP that a container sees for itself is the same that the other containers see for it. Aside from being simpler, it also enables applications to be ported rather easily from virtual machines to containers, since they do not have to change the way they work network-wise.

There are many different networking options that offer these capabilities for Kubernetes: Contiv, Flannel, Nuage Networks, OpenVSwitch, OVN, Project Calico, Romana and Weave Net. For this project, we will use the combination of two of these options: Calico and Flannel, or Canal.

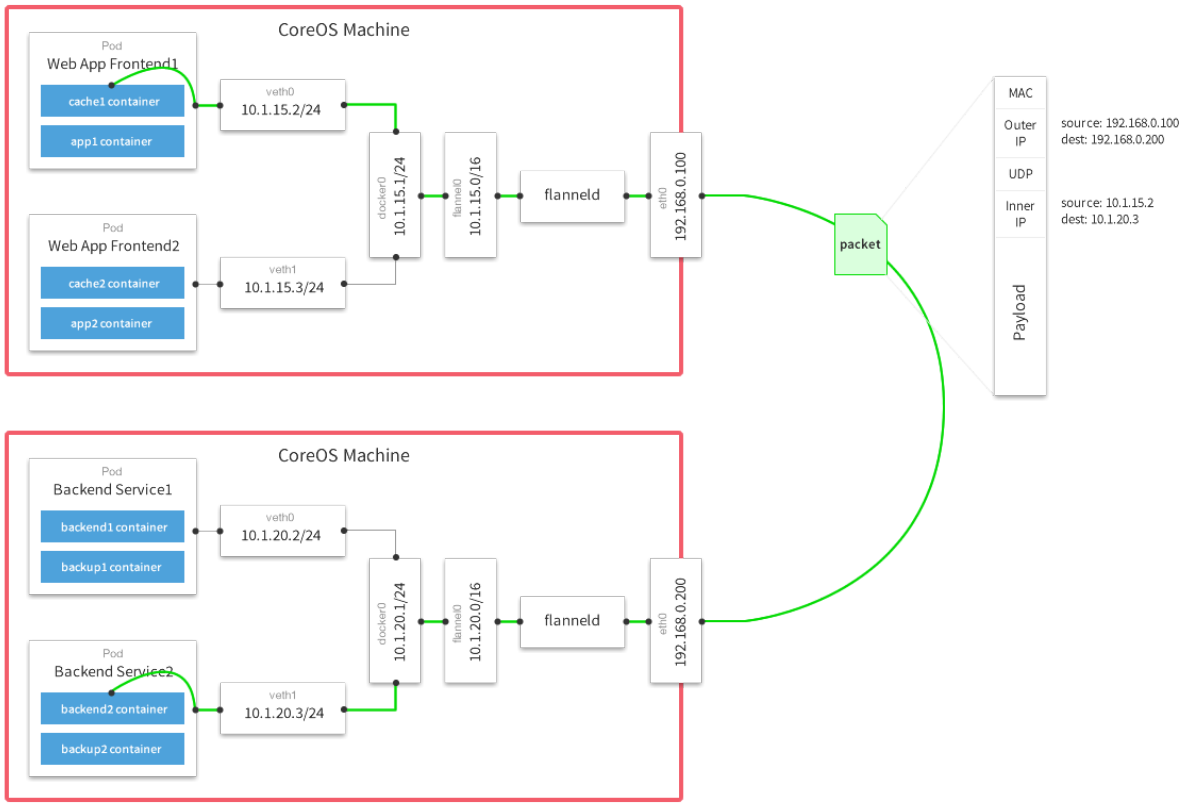
Show me the Canal!

Alright. Let’s talk about Flannel and Calico then.

Calico's and flannel (these are the real logos, I'm not just putting random images here)

Flannel allows inter-pod communication between different hosts by providing an overlay software-defined network (SDN). This solves the main issue we had the Docker networking model. As I said before, when using Docker, each container has an IP address that allows it to communicate with other containers **on the same host**. When pods are placed in different hosts, they rely on their host IP address. Therefore, communication between them is possible by port-mapping. This is fine at a container-level, but applications running inside these containers can have a hard time if they need to advertise their external IP and port to everyone else.

Flannel helps by giving each host a different IP subnet range. The Docker daemon will then assign IPs from this range to containers. Then containers can talk to each user using these unique IP addresses by means of packet encapsulation. Imagine that you have two containers, Container A and Container B. Container A is placed on Host Machine A, and Container B is placed on Host Machine B. When Container A wants to talk to Container B, it will use container B's IP address as the destination address of his packet. This packet will then be encapsulated with an outer UDP packet between Host Machine A and Host Machine B, which will be sent by Host Machine A, and that will have Host Machine B's IP address as the destination address. Once the packet arrives to Host Machine B, the encapsulation is removed and the packet is routed to the container using the inner IP address. The flannel configuration regarding the container/Host Machine mapping is stored in etcd. The routing is done by a flannel daemon called flanneld.



Like this, see? [1]

Calico secures this overlay network, restricting traffic between the pods based on a fine-grained network policy. As I said before, the default Kubernetes behaviour is to allow traffic from all sources inside or outside the cluster to all pods within the cluster. Little reminder from the Kubernetes networking model:

- **all containers can communicate with all other containers without NAT**
- **all nodes can communicate with all containers (and vice-versa) without NAT**
- **the IP that a container sees itself as is the same IP that others see it as**

For security and multi-tenancy reasons, it is coherent to restrict communication between sets of pods on the Kubernetes cluster. Calico supports the v1alpha1 network policy API for Kubernetes. Basically what it does is that it enables network isolation to limit connectivity from an optional set of sources to an optional set of destination TPC/UDP ports. This does not limit the access to the pods by the host itself, as it is necessary for Kubernetes health checks.

With that in mind, inter-pod communication can be restricted at a namespace level, or using particular network policies, using selectors to select the concerned nodes.

I chose Flannel for the SDN part because it is the standard SDN tool for CoreOS (Container Linux), it is shipped with the distribution, it is rather easy to configure, and the documentation is great. I chose Calico because I wanted to use test policy-based security management on Kubernetes, and because of its tight integration with Flannel. They both rely on etcd, which

is rather cool since I'm running an etcd cluster anyways (Calico can be used without an etcd cluster, using the Kubernetes API as a datastore, but this is still experimental).

By the way, Calico can be used as a standalone component, that will handle both the SDN and the network policy-based security management. Then again, Flannel has a few additional networking options, such as udp, vxlan, and even AWS VPC route programming (in case you ever need it).

Ok, now I get it. So, how did you do it?

I think I have to talk to you about the way I see Infrastructure as Code, and explain the tools of the trade first.

Infrastructure as code

Whenever I think about automating the creation of a platform or an application using Infrastructure as Code, I think about three different stages: provisioning, configuration and deployment.

Provisioning is the process of creating the virtual resources in which your application or platform will run. The complexity of the resources may vary depending on your platform: from simple virtual machines if you're working locally, to something slightly more elaborate if you're working on the cloud (network resources, firewalls, and various other services).

Configuration is the part in which you configure your virtual machines so that they can behave in a certain way. This stage includes general OS configuration, security hardening, middleware installation, middleware-specific configuration, and so on.

Deployment is usually application deployment, or where you put your artefacts in the right place in order to make your applications work on the previously configured resources.

Sometimes, a single tool for all these stages will do. Sometimes, it will not. Most of the time I try to keep my code as simple and replaceable as possible (good code is easy to delete, right?). Don't get me wrong, I won't play on hard mode or try to do everything using shell scripts. I'll just try to keep it simple.

But let me talk to you a bit about CoreOS first.

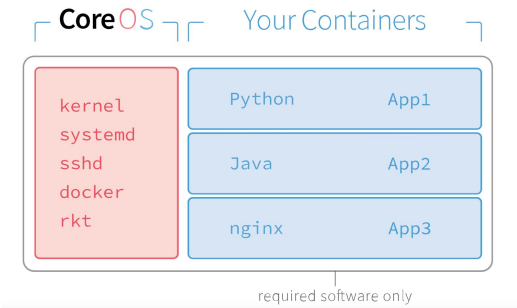
Less is more: Enter CoreOS (yeah yeah, Container Linux)

Container Linux by CoreOS (formerly known as CoreOS Linux, or just CoreOS) is a **lightweight Linux distribution** that uses containers to run applications. This is a game changer if you're used to standard Linux distributions, in which you install packages using a package manager. This thing doesn't even have a package manager.

```
CoreOS stable (1185.5.0)
core@etcd-01 ~ $ sudo yum install vim
sudo: yum: command not found
core@etcd-01 ~ $ sudo apt-get install vim
sudo: apt-get: command not found
core@etcd-01 ~ $ sudo pacman -S vim
sudo: pacman: command not found
core@etcd-01 ~ $ sudo emerge --ask app-editors/vim
sudo: emerge: command not found
core@etcd-01 ~ $
```

Now what?!

It just ships with the basic GNU Core Utilities so you can move around, and then some tools that will come in handy for our quest. These include Kubelet, Docker, etcd and flannel. I'll explain how these things work and how I'm going to use them in the whole Kubernetes journey later. Just keep in mind that CoreOS (yeah yeah, Container Linux) is an OS specifically designed to run containers, and that we're going to take advantage of that in our context.



CoreOS as a distribution [1]

So we're going to manually create these CoreOS virtual machines before installing Kubernetes, right?

Well, no, not really.

Provisioning: Vagrant to the rescue

Vagrant is pretty rad too. It allows you to create reproducible environments based on virtual machines on many different backends, using code. So you just write something called a Vagrantfile, in which you specify all the machines you want and their configuration using Ruby. Then, you type `vagrant up` and all your virtual machines will start popping up.

For this we're using VirtualBox as a provider. There are many others, in case you're feeling creative.

So, once we have all the virtual hosts we need running in our computer, we're just going to manually configure everything in them, right?

Well, no, not really.

Configuration and Deployment: Ansible zen

You do know Ansible, right? Just so you know, an **ansible** is a category of fictional device or technology capable of instantaneous or superluminal communication. The term was first used in Rocannon's World, a science fiction novel by Ursula K. Le Guin. Oh, it is also an IT automation tool.

I really like Ansible because *most of the time**, the only thing you need in order to use it is a control machine (which can be the same computer you're using to code) and SSH access to the target hosts. No complex architectures or master-slave architectures. You can start coding right away!

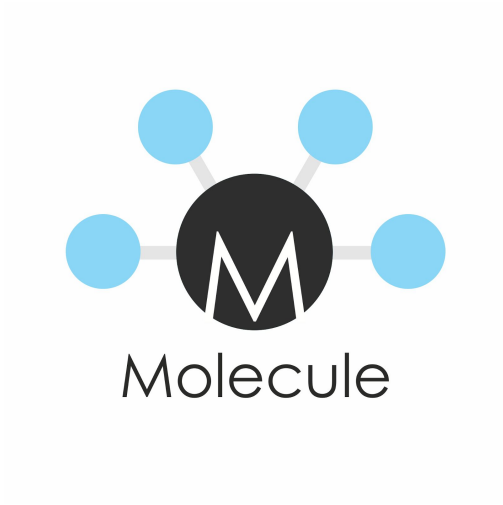
**: This is not one of those cases, but we'll get to that in the next part.*

So, we can configure our platform automatically using Ansible. We're just going to create our machines automatically, configure our resources automatically, and just hope it works, right?

Well, no, not really.

Test and conquer: Molecule

Molecule is a testing tool for Ansible code. It spins up ephemeral infrastructure, it test your roles on the newly created infrastructure, and then it destroys the infrastructure. It also checks for a whole range of other things, like syntax, code quality and impotence, so it's pretty well adapted for what we're trying to do here.



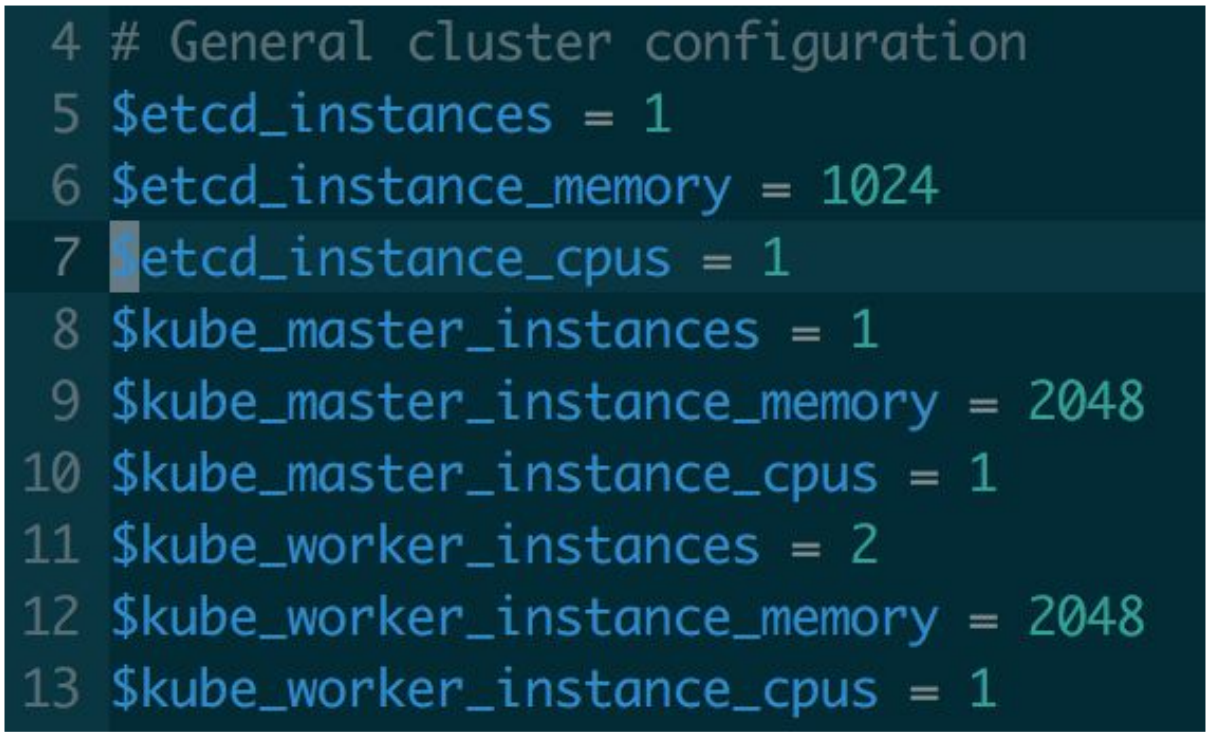
It's an actual molecule!

There are 3 main Molecule drivers: Docker, OpenStack and Vagrant. I usually use the Docker driver for testing roles, due to the fact that a container is usually lightweight, easy to spin up and destroy, and faster than a virtual machine. The thing is that it's hard to create a CoreOS container, in order to install Kubernetes to create more containers. Like, I heard you like containers so let me put a container inside your container so you can schedule containers inside containers while you schedule containers inside containers. Besides, there are no CoreOS Docker images as of this moment. Therefore, we'll be using the Vagrant driver. The target platform just happens to be Vagrant and VirtualBox. Huh.

So we're going to test our code on VirtualBox virtual machines launched by Vagrant, which is exactly the platform we're using for our project. Great.

Excellent, but just show me the code!

Right, code. The first step is to actually create the CoreOS virtual machines. I used a really simple Vagrantfile in which I specify how many instances I want, and how much computing resources each one of them is going to have:



Amazing complexity

You can modify the amount of instances you want to create in this part. Be aware that if you add hosts, you will also need to add them in the inventory for the Ansible code to target all the machines.

I also created a simple IP addressing plan. The principle is that each machine subtype is going to have less than 10 nodes. So I just number them from 10.0.0.1x1 to 10.0.0.1x9, with x being 0 for the etcd nodes, 1 for the Kubernetes master nodes, and 2 for the Kubernetes worker nodes:

```
42  # Kubernetes Master instances configuration
43  (1..$kube_master_instances).each do |i|
44    config.vm.define vm_name = "kube-master-%02d" % i do |config|
45      # Name
46      config.vm.hostname = vm_name
47
48      # RAM, CPU
49      config.vm.provider :virtualbox do |vb|
50        vb.gui = false
51        vb.memory = $kube_master_instance_memory
52        vb.cpus = $kube_master_instance_cpus
53      end
54
55      # IP
56      config.vm.network :private_network, ip: "10.0.0.#{i+110}"
57    end
58  end
```

Absolute power

By the way, you can configure your virtual machines by specifying an Ansible playbook, using the Ansible provisioner in your Vagrantfile like this. I choose not to do so because I like having my `vagrant up` actions separate from my actual configuration. Extra coupling does not abide by this philosophy.

Now you're only a `vagrant up` away from having your CoreOS virtual machines running on your computer. After that, you can export the SSH configuration used by Vagrant with `vagrant ssh-config > ssh.config`. You can then use this configuration for the Ansible configuration, if you include it in your ansible.cfg file:

```
6 [ssh_connection]
7 ssh_args = -F ssh.config
```

Like this

I really like Makefiles. I use them quite often when I have write more than one long command, or many different ones that are going to take a while. I'm also kinda lazy. So I just did this:

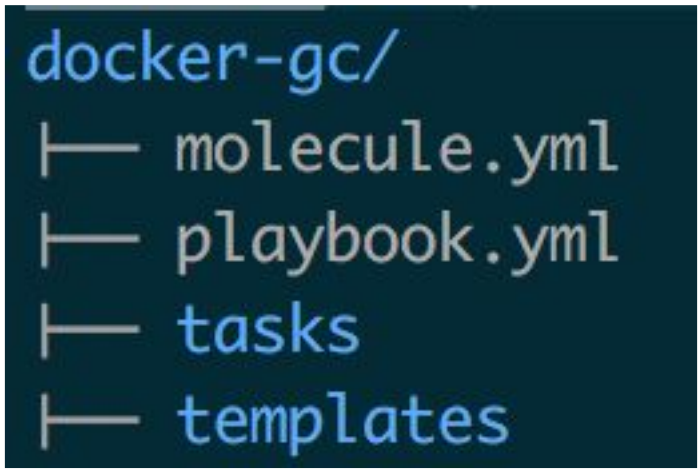
```
5 vagrant:
6 ▸       @vagrant up
7 ▸       @vagrant ssh-config > ssh.config
```

This way, `vagrant up && vagrant ssh-config > ssh.config` becomes `make vagrant`, which saves me 3,7 seconds every day.

You can check the Vagrantfile, the ansigle.cfg and the Makefile on the GitHub repo.

Let's set up our testing configuration now.

Molecule uses a yaml configuration file called molecule.yml. You can use it to define the configuration of your test infrastructure, and to specify the playbook that you want to test. Usually, I try to test each role independently. That means that I'll have a molecule.yml file per role directory, and also a playbook.yml file that uses the role that I'm testing.



Like this

The molecule.yml file specifies an infrastructure (I usually use a single Docker container), and it also specifies which playbook needs to be used on the previously defined infrastructure. For the example in the image above, the playbook.yml file would just include the docker-gc role. Molecule also does this for you automatically if you type `molecule init --driver docker` inside of your role directory. Dope.

This is a pretty good testing strategy when you have roles that are independent from each other. It works like a charm when I test my docker-gc role. The thing is that for this project, the different components depend on each other. I cannot test that my kube-master role is working if I don't have an etcd cluster running. I cannot test that my kube-worker role is working if I don't have at least one Kubernetes master node. Tricky.

So instead of creating a molecule.yml file for each role, we're going to create one at the root of the project, that is going to test our master playbook. Inside of it, we're going to specify a sample test architecture, which is going to be the same one we defined in our Vagrantfile. We're also going to specify the playbook that we want to test. We're going to name it kubernetes.yml (creative right?). You can see the molecule.yml file here.

You can test the roles using the `molecule test` command. This will:

- Create up the infrastructure (create)
- Verify the syntax of the Ansible roles (syntax)
- Execute the playbook (converge)
- Check that the playbook is idempotent by checking for the diffs that a new execution would apply with a dry run (idempotence)
- Destroy the previously created infrastructure (destroy)

These actions can be ran separately by typing `molecule <action>`, and replacing action with one of the expressions between parentheses above. For example, `molecule converge` will just play your playbook on the hosts. You get the idea, right?

I added the `molecule test` command to my Makefile, under the `test` target. That means that I can run `make test` instead of `molecule test`. That makes me gain 0.4 seconds per day. We're up to 4.1 seconds in total. Sweet!

Now that we got our virtual machines running, and our testing configuration ready, let's start configuring stuff. We just gotta take care of a little problem before we can go YOLO with Ansible though.

CoreOS and Ansible

Remember back then when I said that CoreOS ships only with the basics? That means no Python. By extension, that means no Ansible. That's why this time it's a little bit more tricky.

So we need to do something about that. If CoreOS won't come to Python, Python must go to CoreOS.

Since we cannot use Ansible to install Python because we need Python to execute Ansible modules in the first place, we'll just install Python manually in each one of the machines, right?

Well, no, not really.

By the way, if the previous question made you think about the chicken or the egg causality dilemma, just know that the egg came first.

There are three Ansible modules that do not need Python installed on the target host: the raw module, the script module, and the synchronize module. The first allows you to execute an SSH command, without going through the module subsystem. The second one allows you to copy a script to a remote host, and execute it using the remote host's shell environment. The third is a wrapper around rsync, which just uses rsync on the control machine and the remote host.

Using these modules, we can install a lightweight Python implementation called PyPy. The workflow is as follows: I verify that Python is installed using the raw module, and if that is not the case, I install it using a more raw tasks.

```
- block:
  - name: Check if Python is installed
    raw: "{{ ansible_python_interpreter }} --version"
    register: python_install
    changed_when: false

  - name: Check if install tar file exists
    raw: "stat /tmp/pypy-{{ pypy_version }}.tar.bz2"
    register: pypy_tar_file
    changed_when: false

  - name: Check if pypy directory exists
    raw: "stat {{ pypy_dir }}"
    register: pypy_directory
    changed_when: false

  - name: Check if libtinfo is simlinked
    raw: "stat {{ pypy_dir }}/lib/libtinfo.so.5"
    register: libtinfo_symlink
    changed_when: false
  ignore_errors: yes

- name: Download PyPy
  raw: wget -O /tmp/pypy-{{ pypy_version }}.tar.bz2 https://bitbucket.org/pypy/pypy/downloads/pypy-{{ pypy_version }}-linux64.tar.bz2
  when: pypy_tar_file | failed
```

Right on

Why use the `changed_when: false` flag on the tasks inside the block, you say? The thing is that each time you execute a raw task, a change will be made, since the shell command is actually being executed on the target host. That means that each time you run your playbook, you will execute the tasks, no matter what. So if you're downloading things, creating directories, or adding lines to configuration files, you will do so multiple times. This is not idempotent. That's why I verify the state of the Python installation before installing Python, and only execute it when Python is not installed. I just add the `changed_when: false` flag to the verification tasks, since they only verify the existence of the Python associated resources; there are no system modifications because of them.

I feel this is slightly better than executing a shell script with every task embedded into it. It allows me to replay tasks when certain script do not exist, and to have a clear idea of what failed when I get an error: I know right away which task failed, which helps in the debugging process.

> **Note**
> It is usually preferable to write Ansible modules than pushing scripts. Convert your script to an Ansible module for bonus points!

Thanks mom [1]

I did not create this approach, by the way. I just optimised it to make it actually idempotent. I guess you need to stand in the shoulders of giants in order to further from time to time, right?

Let us revisit testing for a second. As a general rule when writing Ansible code, I try to tag roles and tasks as much as possible. This helps a lot if you want to execute only one part of your playbook, or only one role. I also try to use smoke tests whenever it is possible. This means that I'll check that the main feature of my role is working after executing it. If I'm

installing Python, I'll just do something like `python --version` and check that I don't get any errors. For that, inside of each role's tasks directory I'll try to create a main.yml file, which will in turn include a configure.yml file and a test.yml file. The configure.yml file will do all the installation/configuration of the specified component, and the test.yml file (tagged with the test tag) that will test the component, if possible.

```
4 - name: Install and configure PyPy
5   include: configure.yml
6
7 - name: Test PyPy installation
8   include: test.yml
9   tags: [ test ]
```

Smoke test all the way!

By doing this, you will actually test that your infrastructure is running and that it is probably properly configured. Then, if you want to run nothing but your tests, you can do it if you specify the `test` tag while running a playbook. Something like `ansible-playbook -i inventories/vagrant.ini kubernetes.yml --tags test`.

And thus, the 'smoketest' target on my Makefile is born.

Let us continue.

SSL

I won't go really deep into this part. OpenSSL exists since 1998, so it's not exactly recent news. I create a CA for the whole cluster, and then create keys and sign certificates for the API server, for each one of the workers, and for the administrator (the one that's going to be used by you when configuring kubectl).

etcd

In this deployment we're using a single etcd node. You can modify the number of instances from the Vagrantfile, the Ansible code is able to handle a multi-node cluster. Just use odd numbers, because of fault tolerance.

Anyways, configuration is pretty straightforward on the single-node scenario. I just configure it to listen on every interface, and then add the advertise client url to the etcd unit using environment variables:

```
# {{ ansible_managed }}
[Service]
{% if groups ['etcd'] | length == 1 %}
Environment=ETCD_LISTEN_CLIENT_URLS=http://0.0.0.0:2379
Environment=ETCD_ADVERTISE_CLIENT_URLS=http://{{ ansible_env.COREOS_PUBLIC_IPV4 }}:2379
```

Straightforward

And it gets slightly trickier with a multi-node configuration, since the nodes need to be aware of each other, using the ETCD_INITIAL_CLUSTER variable. You also need to provide a node name, and a cluster token for everything to work. There are other options, like using an existing etcd cluster as a discovery mechanism (but we don't have one at the moment), or a public etcd discovery system.

```
{% else %}
Environment=ETCD_NAME={{ ansible_hostname }}
Environment=ETCD_INITIAL_ADVERTISE_PEER_URLS=http://{{ hostvars[ansible_hostname]['ansible_env']['COREOS_PUBLIC_IPV4'] }}:2380
Environment=ETCD_LISTEN_PEER_URLS=http://{{ hostvars[ansible_hostname]['ansible_env']['COREOS_PUBLIC_IPV4'] }}:2380
Environment=ETCD_LISTEN_CLIENT_URLS=http://{{ hostvars[ansible_hostname]['ansible_env']['COREOS_PUBLIC_IPV4'] }}:2379,http://127.0.0.1:2379
Environment=ETCD_ADVERTISE_CLIENT_URLS=http://{{ hostvars[ansible_hostname]['ansible_env']['COREOS_PUBLIC_IPV4'] }}:2379
Environment=ETCD_INITIAL_CLUSTER_TOKEN=etcd-cluster-1
Environment=ETCD_INITIAL_CLUSTER={% for host in groups['etcd'] %}{{ host }}=http://{{ hostvars[host]['ansible_env']['COREOS_PUBLIC_IPV4'] }}:2380{% if not loop.last %},
{% endif %}{% endfor %}

Environment=ETCD_INITIAL_CLUSTER_STATE=new
{% endif %}
```

Less straightforward

All of these configurations can be made either with environment variables or with flags when starting the etcd2 service. The ansible_env.COREOS_PUBLIC_IPV4 variable will be replaced by the node's public IP. I do this often on this project. Then, I just start and enable the service. This is done with the systemd module, and that's why we need Ansible 2.2.

The test part of the role verifies that machine is listening on port 2379, that the etcd cluster is reachable via etcdctl, and then it verifies that the "coreos.com" default namespace exists. It's a simple, effective smoke test.

With our working 1-node etcd **cluster** (get it?), we'll configure the Kubernetes master node.

Kubernetes master node

So basically, we need to create an API server, a Scheduler, and a Controller Manager Server on the Master node.

First, we'll add the TLS resources needed for the master node. That means the CA certificate and the API server certificate and key. Nothing complicated about that.

Next up, networking. For this we'll use flannel and Calico.

In order to configure flannel, we just add configuration environment variables under `/etc/flannel/options.env`. These specify that the flannel interface is this node's public IP, and that the cluster configuration is stocked in etcd cluster:

```
1 # {{ ansible_managed }}
2 FLANNELD_IFACE={{ ansible_env.COREOS_PUBLIC_IPV4 }}
3 FLANNELD_ETCD_ENDPOINTS={{ etcd_endpoints }}
```

/etc/flannel/options.env

Then, we add a system-drop in (a method for adding or overriding parameters of a systemd unit) for flannel, in which we specify that we want to use the configuration specified above when the service launches:

```
1 # {{ ansible_managed }}
2 [Service]
3 ExecStartPre=/usr/bin/ln -sf /etc/flannel/options.env /run/flannel/options.env
```

/etc/systemd/system/flanneld.service.d/40-ExecStartPre-symlink.conf

System drop-ins are pretty cool because they only modify the specific settings you modified, and everything else stays the same.

The flannel configuration is actually stored in etcd. We create it under the coreos.com/network/config namespace using a simple **uri task**.

After that, we gotta configure Docker on the virtual machine. Actually, the only thing we need is to be sure that flannel is used for networking. Basically, flanneld needs to be running when Docker starts:

```
1 # {{ ansible_managed }}
2 [Unit]
3 Requires=flanneld.service
4 After=flanneld.service
```

/etc/systemd/system/docker.service.d/40-flannel.conf

Now that the basic requirements are configured, we're going to configure a whole set of components that are necessary in order to run a Kubernetes cluster: the kubelet, the Kubernetes Proxy, the Controller manager, and the Scheduler. With the exception of the kubelet, all the other components will be deployed in a container form. This is why I said in the first article that the Master node actually does run some containers. How cool is that?

First up, the kubelet. This is the agent on the node that actually starts and stops containers, and communicates with the Docker engine at a host level. It is present on every node in a Kubernetes cluster: both master and worker nodes. It talks with the API server using the certificates we created earlier. The kubelet **does not** manage containers that are not created by Kubernetes. The master node configuration of the kubelet does not register for cluster work (since it is a master and not a worker).

The kubelet may use different standards for networking. One of these standards is the Container Network Interface, or CNI. The CNI is a set of specifications and libraries for writing plugins to configure network interfaces in Linux containers. The only concern of CNI is network connectivity of these containers, and then removing allocated resources when the containers are deleted.

So when using Calico, the kubelet uses CNI for networking. Calico is aware of each pod that is created, and it allows them into the flannel SDN. Both flannel and Calico communicate using CNI interfaces to ensure that the correct IP range is used for each node.

The kubelet configuration can be seen **here**. As of this moment configuration files start to get really verbose so I'll just mention the most important parts. This one specifies the address of the API server, the network plugin to use, the DNS service address and general kubelet configuration, like log files location and configuration directories. The configuration also creates a "manifests" directory, and watches it. This means that for every Pod manifest that is stored in that location, a matching Pod will be created by the kubelet, just as if the Pod was submitted via the API. We will take advantage of this functionality extensively from now on.

After that, we gotta set up the API server. This stateless server takes in requests, process them and stores the result in etcd. It's one of the main components of the Kubernetes cluster. What we would normally do in order to create a server of this kind is to send a request to the API server, with the Pod manifest of the Pod we want to create. But we don't have an API server yet. Huh.

We're going to use the manifest directory that we mentioned on the previous paragraph. When we place the API server manifest inside of the "manifests" directory, it will be automatically created as soon as the kubelet is launched. Pretty neat, huh? We're going to use this same strategy for the Proxy, the Controller manager and the Scheduler.

The API server configuration is pretty long and might be confusing at first. Between many other things, it needs to:

- Be accessible on the host machine address
- Be able to access etcd
- Be aware of the service range we're going to use for service IPs
- Access the SSL certificates we created earlier

And so on. If you want to take a look at the configuration, the template is right **here**.

Then, there's the Kube Proxy. It redirects traffic directed to specific services and pods to their destination. It talks to the API server frequently. In this case, it is used in order to access the API server from the outside. It's configuration can be found **here**.

Let's take on the Controller manager. This component basically applies the necessary changes based on the Replication Controllers. When you increase or decrease the replica count of a pod, it sends a scale up/down event to the API server, and then new containers are scheduled by the Scheduler. It's configuration can be found **here**.

Last but not least, we need to add the Scheduler configuration. This component watches the API for unscheduled Pods, then he finds a machine for them to run and informs the API server of the best choice.

We haven't configured Calico yet, have we? Let's add it as a service. We will create the "/etc/systemd/system/calico-node.service" file. It's configuration can be found **here**. Basically, it talks with etcd in order to store information. Now every container launched will be able to connect itself to the flannel network with its own IP address, and policies created using the policy API will be enforced. Calico also needs a policy-controller in order to work. This component will watch the API and look for changes in the network policy, in order to implement them on Calico. It's configuration can be found **here**.

Finally, (yes, this time it's true) the kubelet service configuration specified a **cni configuration file** we need to create. This file specifies which CNI plugin needs to be called on startup. This creates the flannel plugin, but then delegates control to the Calico plugin. This might sound convoluted at first, but it is actually done so that Calico knows which IP range to use (which is determined before by flannel). It's a pretty short configuration so I'll just put it here:

Easy

After that, we'll just start the services and cross our fingers (don't worry, it will work).

You can see that this role actually has another yaml file embedded into the main.yml file. It is called namespaces.yml. It is included because we need to create a Kubernetes namespace for the Calico policy-controller to run (we specified that **here**). And that needs to be done after the API server starts responding, since it is a Kubernetes functionality. So we just create the calico-system namespace if it doesn't exist already.

By the way, this Master node is not highly available. In order for to have high availability on the Master node, I would need to add a component in order to manage the virtual IP of the master node, like keepalived, or to handle it with fleet. I might do something about this in the future.

That's all for the master node configuration. We should have a master Kubernetes node running alive and well by now. Are you tired? There's still more!

Kubernetes worker node

The master node does not run any containers, it just handles and manages the cluster. The nodes that actually run the containers are the worker nodes. We're going to configure two of them.

We'll just start by configuring the SSL resources the same way we did it on the Master node. Nothing new here. The code just puts them under the "/etc/kubernetes/ssl" directory. Moving on.

Network-wise, we'll use flannel the same way we did on the Master node. I didn't create a flannel role because I figured that the flannel configuration might change from Master to Worker in the next Kubernetes release (turns out it did with 1.5!). Same with the Docker configuration. We just want flannel to be running before we run Docker.

Next up, the kubelet. We will not disable the `register for cluster` work flag, since we want for these nodes to do the heavy lifting. We will also configure it to talk to the master node, to use the CNI network plugin, the specified DNS server, and its own advertising IP. **Here's** the configuration if you want to check it out.

We're going to tell the kubelet to call flannel as a CNI plugin, and then to delegate the control to Calico, the same way we did it on the master node. We'll need to specify the master node's IP on the configuration here instead of **localhost**, since the the configuration needs to access the API server. The configuration template can be seen **here**.

As we said before, there's a kube-proxy instance in each node. That means there's one on the worker nodes too. It's configuration specifies the master nod. Nothing fancy. It's configuration can be found **here**.

We're going to add a kube-config configuration, in order to specify the TLS resources that are necessary for secure communication between the different Kubernetes components. It can be seen **here**.

Finally, we need a Calico Node Container too, that will fulfil the same role it did on the Master node. It's configuration can be found **here**. After that, we start every service, and we should have our full Kubernetes up and running. It might take a while, depending on your internet connection.

Let's recap for a second. What just happened?

We created one etcd node (if you didn't touch the defaults), then we spawned one Kubernetes master node that uses the etcd cluster to persist the cluster state, and two Kubernetes worker nodes that can host container workloads. Yay!

We still need a way of interacting with the cluster. For that we'll use the standard Kubernetes management tool, kubectl.

Configuring kubectl

You can download kubectl and put it wherever you want in your computer. Try to add it to a directory that's in your PATH, so that you don't have to reference the whole path every time you do a kubectl action. Make sure it is executable, too.

After that you can configure it by specifying the certificates and the Master host's URL:

```
- name: Set default cluster
  command: kubectl config set-cluster default-cluster --server=https://{{ master_host }} --certificate-authority=ca.pem
  args:
    chdir: "{{ kube_resource_dir }}/ca"
  when: "'Kubernetes master' not in cluster_info.stdout"

- name: Set credentials
  command: kubectl config set-credentials default-admin --certificate-authority=ca.pem --client-key=admin-key.pem --client-certificate=admin.pem
  args:
    chdir: "{{ kube_resource_dir }}/ca"
  when: "'Kubernetes master' not in cluster_info.stdout"

- name: Set context
  command: kubectl config set-context default-system --cluster=default-cluster --user=default-admin
  args:
    chdir: "{{ kube_resource_dir }}/ca"
  when: "'Kubernetes master' not in cluster_info.stdout"

- name: Use context
  command: kubectl config use-context default-system
  args:
    chdir: "{{ kube_resource_dir }}/ca"
  when: "'Kubernetes master' not in cluster_info.stdout"
```

There is no kube

This configuration is only applied when the cluster is not configured, though. That way, we keep our idempotence neat and clean.

That was easy, kinda. Now what? Oh yeah, cool stuff on our cluster.

Add-ons

Add-ons are... well... things you add-on to your Kubernetes cluster in order to have improved functionality. They are created using Kuberntetes native resources. Most of the time they will be pods, so we can juste create a manifest for them and then create them using the API server (through kubectl).

There are two add-ons that are commonly used on almost every standard Kubernetes installation: the DNS add-on, and the Kubernetes Dashboard add-on.

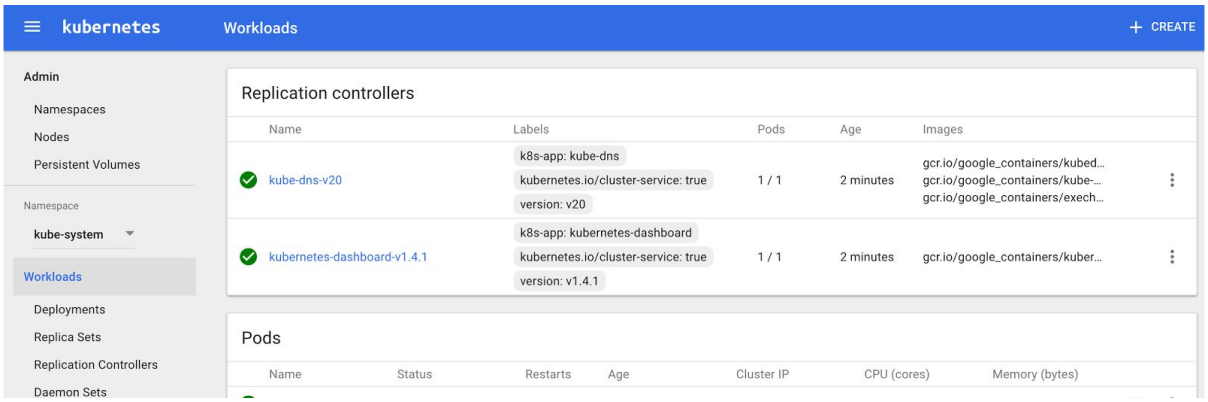
The first enables service discovery for your containers. They can have a DNS name, and they can be reached by other containers with it. The manifest is huge. It's because we're creating a Service, which provides DNS lookups over port 53 to every resource that demands it, and also a replication controller, which makes sure that there is one replica of the pod at all times. The configuration can be seen **here**.

The Dashboard allows you to see general information about your cluster. That means Pods, Services, Replication Controllers, and all that under different namespaces. It's a pretty cool tool, honestly. We'll create a Service and a Replication Controller for it too.

You can access the Kubernetes Dashboard by using the port forwarding functionality of kubectl:

```
kubectl get pods --namespace=kube-system
kubectl port-forward kubernetes-dashboard-v.1.4.1-ID 9090 --namespace=kube-sytem
```

And now, your Kubernetes Dashboard should be accessible on port 9090.



Show me the money!

Woot! Almost done. Now we only need to test that it works, and that our code is working as intended.

So, `molecule test` says that:

```
--> Destroying instances...
--> Checking playbook's syntax...

playbook: kubernetes.yml
--> Creating instances...
Bringing machine 'etcd-01' up with 'virtualbox' provider...
Bringing machine 'kube-master-01' up with 'virtualbox' provider...
Bringing machine 'kube-worker-01' up with 'virtualbox' provider...
Bringing machine 'kube-worker-02' up with 'virtualbox' provider...
==> etcd-01: Cloning VM...
==> etcd-01: Matching MAC address for NAT networking...
```

Our infrastructure is created without any hiccups.

```
--> Starting Ansible Run...

PLAY [Bootstrap coreos hosts] ***************************************************

TASK [bootstrap/ansible-bootstrap : Check if Python is installed] **************
fatal: [etcd-01]: FAILED! => {"changed": false, "failed": true, "rc": 127, "stderr
osed.\r\n", "stdout": "/bin/sh: /home/core/bin/python: No such file or directory\r
...ignoring
```

The playbook runs as intended.

```
PLAY RECAP *********************************************************************
etcd-01                    : ok=44   changed=12   unreachable=0    failed=0
kube-master-01             : ok=54   changed=25   unreachable=0    failed=0
kube-worker-01             : ok=47   changed=23   unreachable=0    failed=0
kube-worker-02             : ok=47   changed=23   unreachable=0    failed=0

--> Idempotence test in progress (can take a few minutes)...
--> Starting Ansible Run...
Idempotence test passed.
--> Executing ansible-lint...
--> Destroying instances...
==> kube-worker-02: Forcing shutdown of VM...
==> kube-worker-02: Destroying VM and associated drives...
==> kube-worker-01: Forcing shutdown of VM...
==> kube-worker-01: Destroying VM and associated drives...
==> kube-master-01: Forcing shutdown of VM...
==> kube-master-01: Destroying VM and associated drives...
==> etcd-01: Forcing shutdown of VM...
==> etcd-01: Destroying VM and associated drives...
```

And that our code is properly linted, and it is idempotent as well!

Let's have some fun with our cluster now.

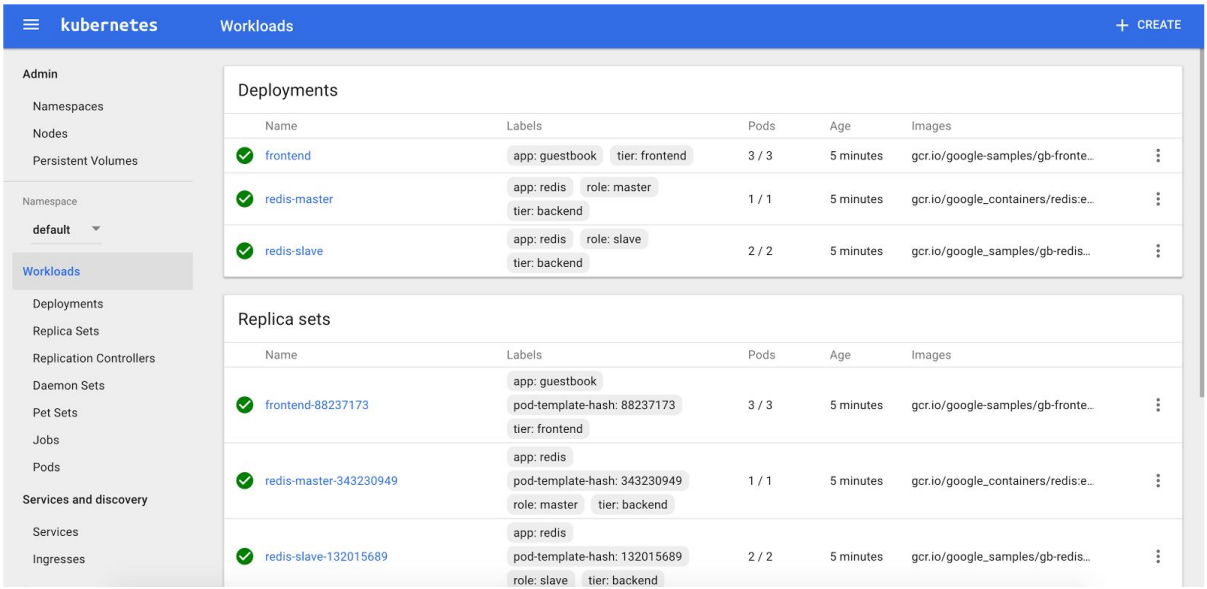
Sample app

We're going to use the guestbook example that's included on the Kubernetes samples. This is a good application to test a proper installation, since it's a multi-tier application, with multiple components speaking to each other, on different nodes.

The guestbook application is created under the kubernetes-resources directory. It can be launched using kubectl:

```
kubectl create -f guestbook.yml
```

We can see that the resources are properly created on the Dashboard:

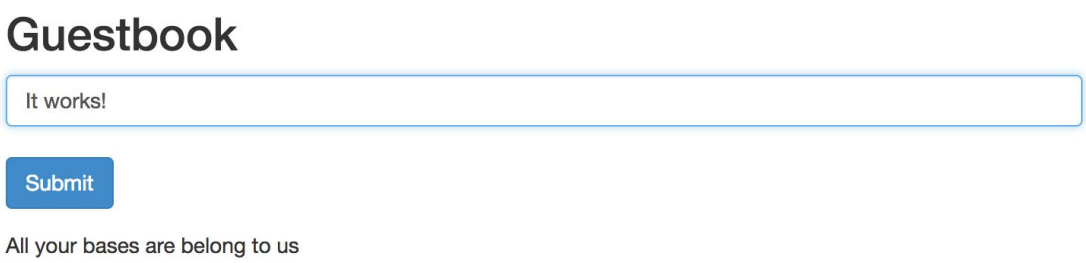


The gang's all here!

And then we can even test the application by port-forwarding to the frontend application:

```
kubectl get pods
kubectl port-forward frontend-ID 8080:80
```

The application should be accessible on port 8080. You can test it by adding a message to the guestbook:



Get it?

Great. It works. Mission accomplished!

So what did we learn in the end?

- CoreOS (Container Linux) is a lightweight Linux distribution that runs almost everything inside of containers
- An "Ansible" is a fictional device capable of instantaneous or superluminal communication. It is also a pretty powerful IT automation tool.
- You can move around on remote hosts using Ansible, even when you don't have Python installed on them
- etcd is a distributed key-value store, which is used as the standard distributed storage by Kubernetes
- Flannel and Calico can be used together to provide SDN-based connectivity and network policies for containers located on different hosts
- You can use Molecule to continuously test important aspects of your Ansible code
- The egg came before the chicken

Final thoughts

Phew, yeah, that was kinda long. Anyways, I hope you had fun. I know I did. Still, I would have loved to work on some other things too:

- Use etcd3 instead of etcd2
- Install Python “The CoreOS way”. The PyPy installation works, but it feels kinda hacky. I’d have loved to run Python from inside a container and provision the host with Ansible somehow
- Use SSL on the etcd cluster
- Use fleet for certain tasks, like deploying unit files, or handling the Master node high availability

I might work on these things in the future. Anyways, see you on the next adventure!

Seb