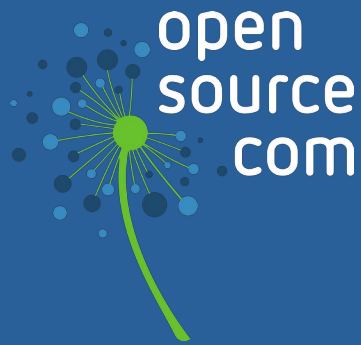




Orchestration with Kubernetes

We are Opensource.com

Opensource.com is a community website publishing stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Do you have an open source story to tell? Submit a story idea at opensource.com/story

Email us at open@opensource.com



Supported by
Red Hat

Table of Contents

What's the difference between orchestration and automation?.....	4
Getting started with Minikube: Kubernetes on your laptop.....	7
Getting started with Kubernetes.....	10
Automate your container orchestration with Ansible modules for Kubernetes.....	15
A beginner's guide to Kubernetes Jobs and CronJobs.....	21
A beginner's guide to Kubernetes container orchestration.....	28
A beginner's guide to load balancing.....	34
A guide to Kubernetes architecture.....	38
Getting started with OKD.....	43

What's the difference between orchestration and automation?

By Seth Kenlon

For the longest time, it seemed the only thing any sysadmin cared about was automation. Recently, though, the mantra seems to have changed from automation to orchestration, leading many puzzled admins to wonder: "What's the difference?"

The difference between automation and orchestration is primarily in intent and tooling. Technically, automation can be considered a subset of orchestration. While orchestration suggests many moving parts, automation usually refers to a singular task or a small number of strongly related tasks. Orchestration works at a higher level and is expected to make decisions based on changing conditions and requirements.

However, this view shouldn't be taken too literally because both terms—*automation* and *orchestration*—do have implications when they're used. The results of both are functionally the same: things happen without your direct intervention. But the way you get to those results, and the tools you use to make them happen, are different, or at least the terms are used differently depending on what tools you've used.

For instance, automation usually involves scripting, often in Bash or Python or similar, and it often suggests scheduling something to happen at either a precise time or upon a specific event. However, orchestration often begins with an application that's purpose-built for a set of tasks that may happen irregularly, on demand, or as a result of any number of trigger events, and the exact results may even depend on a variety of conditions.

Decisionmaking and IT orchestration

Automation suggests that a sysadmin has invented a system to cause a computer to do something that would normally have to be done manually. In automation, the sysadmin has

already made most of the decisions on what needs to be done, and all the computer must do is execute a "recipe" of tasks.

Orchestration suggests that a sysadmin has set up a system to do something on its own based on a set of rules, parameters, and observations. In orchestration, the sysadmin knows the desired end result but leaves it up to the computer to decide what to do.

Consider Ansible and Bash. Bash is a popular shell and scripting language used by sysadmins to accomplish practically everything they do during a given workday. Automating with Bash is straightforward: Instead of typing commands into an interactive session, you type them into a text document and save the file as a shell script. Bash runs the shell script, executing each command in succession. There's room for some conditional decisionmaking, but usually, it's no more complex than simple if-then statements, each of which must be coded into the script.

Ansible, on the other hand, uses playbooks in which a sysadmin describes the desired state of the computer. It lists requirements that must be met before Ansible can consider the job done. When Ansible runs, it takes action based on the current state of the computer compared to the desired state, based on the computer's operating system, and so on. A playbook doesn't contain specific commands, instead leaving those decisions up to Ansible itself.

Of course, it's particularly revealing that Ansible is referred to as an automation—not an orchestration—tool. The difference can be subtle, and the terms definitely overlap.

Orchestration and the cloud

Say you need to convert a file type that's regularly uploaded to your server by your users.

The manual solution would be to check a directory for uploaded content every morning, open the file, and then save it in a different format. This solution is slow, inefficient, and probably could happen only once every 24 hours because you're a busy person.

You could automate the task. Were you to do that, you might write a PHP or a Node.js script to detect when a file has been uploaded. The script would perform the conversion and send an alert or make a log entry to confirm the conversion was successful. You could improve the script over time to allow users to interact with the upload and conversion process.

Were you to orchestrate the process, you might instead start with an application. Your custom app would be designed to accept and convert files. You might run the application in a container on your cloud, and using OpenShift, you could launch additional instances of your app when the traffic or workload increases beyond a certain threshold.

Learning automation and orchestration

There isn't just one discipline for automation or orchestration. These are broad practices that are applied to many different tasks across many different industries. The first step to learning, though, is to become proficient with the technology you're meant to orchestrate and automate. It's difficult to orchestrate (safely) the scaling a series of web servers if you don't understand how a web server works, or what ports need to be open or closed, or what a port is. In practice, you may not be the person opening ports or configuring the server; you could be tasked with administering OpenShift without really knowing or caring what's inside a container. But basic concepts are important because they broadly apply to usability, troubleshooting, and security.

You also need to get familiar with the most common tools of the orchestration and automation world. Learn some [Bash](#), start using [Git](#) and design some [Git hooks](#), learn some Python, get comfortable with [YAML](#) and [Ansible](#), and try out Minikube, [OKD](#), and [OpenShift](#).

Orchestration and automation are important skills, both to make your work more efficient and as something to bring to your team. Invest in it today, and get twice as much done tomorrow.

Getting started with Minikube: Kubernetes on your laptop

By Bryant Son

Minikube is advertised on the [Hello Minikube](#) tutorial page as a simple way to run Kubernetes for containers.

Prerequisites

1. You have installed either [Podman](#) or [Docker](#).
2. Your computer is an RHEL, CentOS, or Fedora-based workstation.
3. You have [installed a working KVM2 hypervisor](#), such as **libvirt**.
4. If you're using Docker, you must have a working **docker-machine-driver-kvm2**. The following commands installs the driver:

```
$ curl -Lo docker-machine-driver-kvm2 \
https://storage.googleapis.com/minikube/releases/latest/docker-machine-
driver-kvm2
$ chmod +x docker-machine-driver-kvm2 \
&& sudo cp docker-machine-driver-kvm2 /usr/local/bin/ \
&& rm docker-machine-driver-kvm2
```

Download, install, and start Minikube

1. Create a directory for the two files you will download: [minikube](#) and [kubectl](#).
2. Open a terminal window and run the following command to install minikube.

```
$ curl -Lo minikube \
https://storage.googleapis.com/minikube/releases/latest/minikube-linux-
amd64
```

Note that the minikube version (for example, minikube-linux-amd64) may differ based on your computer's specs.

3. Use **chmod** to make it executable.

```
$ chmod +x minikube
```

4. Move the file to the **/usr/local/bin** path so you can run it as a command.

```
$ sudo mv minikube /usr/local/bin
```

5. Install kubectl using the following command (similar to the minikube installation process).

```
$ curl -Lo kubectl \
https://storage.googleapis.com/kubernetes-release/release/$(curl -s \
https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/
linux/amd64/kubectl
```

Use the **curl** command to determine the latest version of Kubernetes.

6. Use **chmod** to make kubectl executable.

```
$ chmod +x kubectl
```

7. Move kubectl to the **/usr/local/bin** path to run it as a command.

```
$ mv kubectl /usr/local/bin
```

8. Run **minikube start**. To do so, you need to have a hypervisor available. I used KVM2, but you can also use Virtualbox. Make sure to run the following command as a user instead of root so the configuration will be stored for the user instead of root.

```
$ minikube start --vm-driver=kvm2
```

It can take quite a while, so wait for it.

9. Minikube should download and start. Use the following command to make sure it was successful.

```
$ cat ~/.kube/config
```

10. Execute the following command to run Minikube as the context. The context is what determines which cluster kubectl is interacting with. You can see all your available contexts in the ~/.kube/config file.

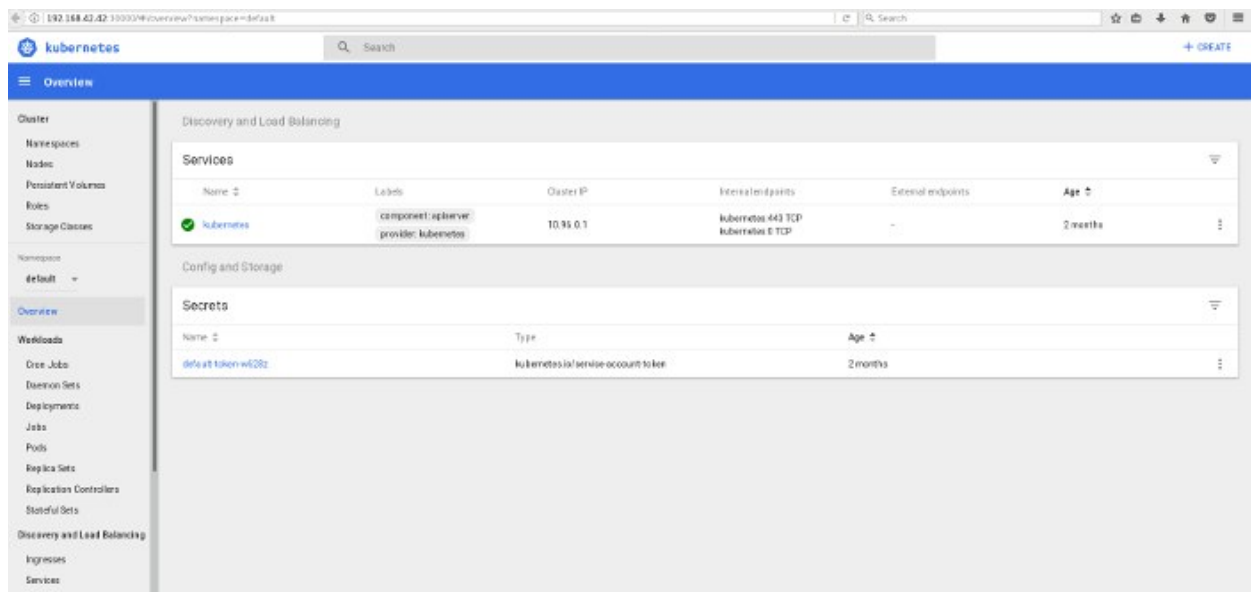
```
$ kubectl config use-context minikube
```

11. Run the **config** file command again to check that context Minikube is there.

```
$ cat ~/.kube/config
```

12. Finally, run the following command to open a browser with the Kubernetes dashboard.

```
$ minikube dashboard
```



Getting started with Kubernetes

By Ben Finkel

Kubernetes allows you to create a portable and scalable application deployment that can be scheduled, managed, and maintained easily. As an open source project, Kubernetes is continually being updated and improved, and it leads the way among container cluster management software.

Kubernetes uses various architectural components to describe the deployments it manages.

- [Pods](#) are a group of one or more containers that share network and storage. The containers in a pod are considered "tightly coupled," and they are managed and deployed as a single unit. If an application were deployed in a more traditional model, the contents of the pod would always be deployed together on the same machine.
- [Nodes](#) represents a worker machine in a Kubernetes cluster. The worker machine can be either physical or (more likely) virtual. A node contains all the required services to host a pod.
- A cluster always requires a [master](#) node, where the controlling services (known as the master components) are installed. These services can be distributed on a single machine or across multiple machines for redundancy. They control communications, workload, and scheduling.
- [Deployments](#) are a way to declaratively set a state for your pods or ReplicaSets (groups of pods to be deployed together). Deployments use a "desired state" format to describe how the deployment should look, and Kubernetes handles the actual deployment tasks. Deployments can be updated, rolled back, scaled, and paused at will.

The following tutorial explains the basics of creating a cluster, deploying an app, and creating a proxy, then send you on your way to learning even more about Kubernetes. In this article, I

assume you're using Minikube for a local instance of Kubernetes. Of course, you're free to use an actual Kubernetes install instead, if you have one available to you.

Create a cluster

In your terminal, start Minikube:

```
$ minikube start
```

View the cluster information with the command:

```
$ kubectl cluster-info
Kubernetes is running at http://example.local:8080 [...]
```

List the available nodes with the command:

```
$ kubectl get nodes
NAME      STATUS   AGE
host01    Ready    22s
```

Your output may differ a little, but as long as you see similar reports, you know you're on the right track.

Deploy an app

In the next step in the interactive tutorial, you'll deploy a containerized application to your cluster with a deployment configuration. This describes how to create instances of your app, and the master will schedule those instances onto nodes in the cluster.

In the interactive terminal, create a new deployment with the **kubectl run** command:

```
$ kubectl run kubernetes-bootcamp \
--image=docker.io/jocatalin/kubernetes-bootcamp:v1 --port=8080
```

This creates a new deployment with the name **kubernetes-bootcamp** from a public repository at docker.io and overrides the default port to 8080.

View the deployment with the command:

```
$ kubectl get deployments
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
kubernetes-bootcamp	1	1	1	1	12s

The deployment is currently on a single node, because only one node is available.

Create a proxy

Now you can create a proxy into your deployed app. A pod runs on an isolated private network that cannot be accessed from outside. The **kubectl** command uses an API to communicate with the application, and a proxy is needed to expose the application for use by other services.

Open a new terminal window and start the proxy server with the command:

```
$ kubectl proxy
Starting to serve on 127.0.0.1:8001
```

This creates a connection between your cluster and the virtual terminal window. Notice it's running on port 8001 on the local host.

Return to the first terminal window and run a **curl** command to see this in action:

```
$ curl http://localhost:8001/version
{ "major": "1",
  "minor": "5",
  "gitVersion": "v1.5.2",
  "gitCommit": "3187edd66737c1265299e1ed47505cd78eb1e6d4",
  "gitTreeState": "clean",
  [...]
}
```

The JSON output displays the version information from the cluster itself.

You can also get a detailed output of your pod by using the command:

```
$ kubectl describe pods
```

This output includes very important information, like the pod name, local IP address, state, and restart count.

Moving forward

Kubernetes is a full-fledged deployment, scheduling, and scaling manager and is capable of deciding all of the myriad details of how to deploy an app on your cluster. The few commands explored here are just the beginning of interacting with and understanding a Kubernetes deployment. The crucial takeaway is how fast and easy it is to do and how few details you need to provide to use Kubernetes.

Follow the online [interactive tutorial](#) to learn more about how Kubernetes works and all that you can do with it.

Automate container orchestration with Ansible modules

By Seth Kenlon

[Ansible](#) is one of the best tools for automating your work. [Kubernetes](#) is one of the best tools for orchestrating containers. What happens when you combine the two? As you might expect, Ansible combined with Kubernetes lets you automate your container orchestration.

Ansible modules

On its own, Ansible is basically just a framework for interpreting YAML files. Its true power comes from its [many modules](#). Modules are what enable you to invoke external applications with just a few simple configuration settings in a playbook.

There are a few modules that deal directly with Kubernetes, and a few that handle related technology like [Podman](#) and [Docker](#). Learning a new module is often similar to learning a new terminal command or a new API. You get familiar with a module from its documentation, you learn what arguments it accepts, and you equate its options to how you might use the application it interfaces with.

Access a Kubernetes cluster

To try out Kubernetes modules in Ansible, you must have access to a Kubernetes cluster. If you don't have that, then you might try to open a trial account online, but most of those are short term. Instead, you can install [Minikube](#), as described on the Kubernetes website or in Bryant Son's article in the book. Minikube provides a local instance of a single-node Kubernetes install, allowing you to configure and interact with it as you would a full cluster.

Before installing Minikube, you must ensure that your environment is ready to serve as a virtualization backend. You may need to install `libvirt` and grant yourself permission to the `libvirt` group:

```
$ sudo dnf install libvirt
$ sudo systemctl start libvirtd
$ sudo usermod --append --groups libvirt `whoami`
$ newgrp libvirt
```

Install Python modules

To prepare for using Kubernetes-related Ansible modules, you should also install a few helper Python modules:

```
$ python3 -m pip install kubernetes --user
$ python3 -m pip install openshift --user
```

Start Kubernetes

If you're using Minikube instead of a Kubernetes cluster, use the `minikube` command to start up a local, miniaturized Kubernetes instance on your computer:

```
$ minikube start --driver=kvm2 --kvm-network default
```

Wait for Minikube to initialize. Depending on your internet connection, this could take several minutes.

Get information about your cluster

Once you've started your cluster successfully, you can get information about it with the `cluster-info` option:

```
$ kubectl cluster-info
Kubernetes master is running at https://192.168.39.190:8443
KubeDNS is running at https://192.168.39.190:8443/api/v1/[...]/kube-dns:dns/proxy
To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

Use the k8s module

The entry point for using Kubernetes through Ansible is the `k8s` module, which enables you to manage Kubernetes objects from your playbooks. This module describes states resulting from

kubectl instructions. For instance, here's how you would create a new [namespace](#) with kubectl:

```
$ kubectl create namespace my-namespace
```

It's a simple action, and the YAML representation of the same result is similarly terse:

```
- hosts: localhost
tasks:
  - name: create namespace
    k8s:
      name: my-namespace
      api_version: v1
      kind: Namespace
      state: present
```

In this case, the host is defined as `localhost`, under the assumption that you're running this against Minikube. Notice that the module in use defines the syntax of the parameters available (such as `api_version` and `kind`).

Before using this playbook, verify it with `yamllint`:

```
$ yamllint example.yaml
```

Correct any errors, and then run the playbook:

```
$ ansible-playbook ./example.yaml
```

Verify that the new namespace has been created:

```
$ kubectl get namespaces
NAME          STATUS    AGE
[...]
my-namespace  Active   3s
```

Pull a container image with Podman

Containers are Linux systems, almost impossibly minimal in scope, that can be managed by Kubernetes. Much of the container specifications have been defined by the [LXC project](#) and Docker. A recent addition to the container toolset is Podman, which is popular because it runs without requiring a daemon.

With Podman, you can pull a container image from a repository, such as Docker Hub or Quay.io. The Ansible syntax for this is simple, and all you need to know is the location of the container, which is available from the repository's website:

```
- name: pull an image
  podman_image:
    name: quay.io/jitesoft/nginx
```

Verify it with `yamllint`:

```
$ yamllint example.yaml
```

And then run the playbook:

```
$ ansible-playbook ./example.yaml
[WARNING]: provided hosts list is empty, only localhost is available.
Note that the implicit localhost does not match 'all'
PLAY [localhost] *****
TASK [Gathering Facts] *****
ok: [localhost]
TASK [create k8s namespace] *****
ok: [localhost]
TASK [pull an image] *****
changed: [localhost]
PLAY RECAP *****
localhost: ok=3 changed=1 unreachable=0 failed=0
           skipped=0 rescued=0 ignored=0
```

Deploy with Ansible

You're not limited to small maintenance tasks with Ansible. Your playbook can interact with Ansible in much the same way a configuration file does with `kubectl`. In fact, in many ways, the YAML you know by using Kubernetes translates to your Ansible plays.

Here's a configuration you might pass directly to `kubectl` to deploy an image (in this example, a web server):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-webserver
spec:
  selector:
    matchLabels:
      run: my-webserver
  replicas: 1
  template:
    metadata:
      labels:
        run: my-webserver
    spec:
      containers:
        - name: my-webserver
          image: nginx
          ports:
            - containerPort: 80
```

If you know these parameters, then you mostly know the parameters required to accomplish the same with Ansible. You can, with very little modification, move that YAML into a **definition** element in your Ansible playbook:

```
- name: deploy a web server
  k8s:
    api_version: v1
    namespace: my-namespace
    definition:
      kind: Deployment
      metadata:
        labels:
          app: nginx
        name: nginx-deploy
      spec:
        replicas: 1
        selector:
          matchLabels:
            app: nginx
        template:
          metadata:
            labels:
              app: nginx
          spec:
```

```
containers:
  - name: my-webserver
    image: quay.io/jitesoft/nginx
    ports:
      - containerPort: 80
        protocol: TCP
```

After running this, you can see the deployment with `kubectl`, as usual:

```
$ kubectl -n my-namespace get pods
NAME                                READY   STATUS
nginx-deploy-7fdc9-t9wc2           1/1     Running
```

Modules for the cloud

As more development and deployments move to the cloud, it's important to understand how to automate the important aspects of your cloud. The `k8s` and `podman_image` modules are only two examples of modules related to Kubernetes and a mere fraction of modules developed for the cloud. Take a look at your workflow, find the tasks you want to track and automate, and see how Ansible can help you do more by doing less.

A beginner's guide to Kubernetes Jobs and CronJobs

By Mike Calizo

[Kubernetes](#) is the default orchestration engine for containers. Its options for controlling and managing pods and containers include:

1. Deployments
2. StatefulSets
3. ReplicaSets

Each of these features has its own purpose, with the common function to ensure that pods run continuously. In failure scenarios, these controllers either restart or reschedule pods to ensure the services in the pods continue running.

As the [Kubernetes documentation explains](#), a Kubernetes Job creates one or more pods and ensures that a specified number of the pods terminates when the task (Job) completes.

Just like in a typical operating system, the ability to perform automated, scheduled jobs without user interaction is important in the Kubernetes world. But [Kubernetes Jobs](#) do more than just run automated jobs, and there are multiple ways to utilize them through:

1. Jobs
2. CronJobs
3. Work queues (this is beyond the scope of this article)

Sounds simple right? Well, maybe. Anyone who works on containers and microservice applications knows that some require services to be transient so that they can do specific tasks for applications or within the Kubernetes clusters.

In this article, I explain why Kubernetes Jobs are important, how to create Jobs and CronJobs, and when to use them for applications running on the Kubernetes cluster.

Differences between Kubernetes Jobs and CronJobs

Kubernetes Jobs are used to create transient pods that perform specific tasks they are assigned to. [CronJobs](#) do the same thing, but they run tasks based on a defined schedule.

Jobs play an important role in Kubernetes, especially for running batch processes or important ad-hoc operations. Jobs differ from other Kubernetes controllers in that they run tasks until completion, rather than managing the desired state such as in Deployments, ReplicaSets, and StatefulSets.

How to create Kubernetes Jobs and CronJobs

With that background in hand, you can start creating Jobs and CronJobs.

Prerequisites

To do this exercise, you need to have the following:

1. A working Kubernetes cluster or a Minikube install
2. The [kubectl](#) Kubernetes command line

Here is the Minikube deployment I used for this demonstration:

```
$ minikube version
minikube version: v1.8.1
$ kubectl cluster-info
Kubernetes master is running at https://172.17.0.59:8443
KubeDNS is running at
https://172.17.0.59:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/
proxy
$ kubectl get nodes
NAME          STATUS    ROLES    AGE   VERSION
minikube      Ready     master   88s   v1.17.3
```

Kubernetes Jobs

Just like anything else in the Kubernetes world, you can create Kubernetes Jobs with a definition file. Create a file called `sample-jobs.yaml` using your favorite editor.

Here is a snippet of the file that you can use to create an example Kubernetes Job:

```
apiVersion: batch/v1      ## The version of the Kubernetes API
kind: Job                 ## The type of object for jobs
metadata:
```

```

name: job-test
spec:          ## What state you desire for the object
template:
  metadata:
    name: job-test
  spec:
    containers:
    - name: job
      image: busybox          ## Image used
      command: ["echo", "job-test"]  ## Command used to create logs for
verification later
      restartPolicy: OnFailure  ## Restart Policy in case container failed

```

Next, apply the Jobs in the cluster:

```
$ kubectl apply -f sample-jobs.yaml
```

Wait a few minutes for the pods to be created. You can view the pod creation's status:

```
$ kubectl get pod -watch
```

After a few seconds, you should see your pod created successfully:

```

$ kubectl get pods
  NAME          READY   STATUS    RESTARTS   AGE
job-test        0/1     Completed   0          11s

```

Once the pods are created, verify the Job's logs:

```
$ kubectl logs job-test job-test
```

You have created your first Kubernetes Job, and you can explore details about it:

```
$ kubectl describe job job-test
```

Clean up the Jobs:

```
$ kubectl delete jobs job-test
```

Kubernetes CronJobs

You can use CronJobs for cluster tasks that need to be executed on a predefined schedule. As the [documentation explains](#), they are useful for periodic and recurring tasks, like running

backups, sending emails, or scheduling individual tasks for a specific time, such as when your cluster is likely to be idle.

As with Jobs, you can create CronJobs via a definition file. Following is a snippet of the CronJob file `cron-test.yaml`. Use this file to create an example CronJob:

```
apiVersion: batch/v1beta1      ## The version of the Kubernetes API
kind: CronJob                  ## The type of object for Cron jobs
metadata:
  name: cron-test
spec:
  schedule: "*/1 * * * *"      ## Defined schedule using the *nix style cron
                                syntax
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: cron-test
              image: busybox      ## Image used
              args:
                - /bin/sh

                - -c
                - date; echo Hello this is Cron test
          restartPolicy: OnFailure ## Restart Policy in case container failed
```

Apply the CronJob to your cluster:

```
$ kubectl apply -f cron-test.yaml
cronjob.batch/cron-test created
```

Verify that the CronJob was created with the schedule in the definition file:

```
$ kubectl get cronjob cron-test
NAME      SCHEDULE      SUSPEND  ACTIVE  LAST SCHEDULE  AGE
cron-test */1 * * * *   False    0       <none>        10s
```

After a few seconds, you can find the pods that the last scheduled job created and view the standard output of one of the pods:

```
$ kubectl logs cron-test-1604870760
Sun Nov  8 21:26:09 UTC 2020
Hello from the Kubernetes cluster
```

You have created a Kubernetes CronJob that creates an object once per execution based on the schedule `schedule: "*/1 * * * *"`. Sometimes the creation can be missed because of environmental issues in the cluster. Therefore, they need to be [idempotent](#).

Other things to know

Unlike deployments and services in Kubernetes, you can't change the same Job configuration file and reapply it at once. When you make changes in the Job configuration file, you must delete the previous Job from the cluster before you apply it.

Generally, creating a Job creates a single pod and performs the given task, as in the example above. But by using completions and [parallelism](#), you can initiate several pods, one after the other.

Use your Jobs

You can use Kubernetes Jobs and CronJobs to manage your containerized applications. Jobs are important in Kubernetes application deployment patterns where you need a communication mechanism along with interactions between pods and the platforms. This may include cases where an application needs a "controller" or a "watcher" to complete tasks or needs to be scheduled to run periodically.

A beginner's guide to Kubernetes container orchestration

By Jiaqi Liu

Last fall, I took on a new role with a team that relies on [Kubernetes](#) (K8s) as part of its core infrastructure. While I have worked with a variety of container orchestrators in my time, the job change sent me back to the basics. Here is my take on the fundamentals you should be familiar with if you're working with Kubernetes.

I consider Kubernetes to have clear advantages when you're working with open source software. As an open source platform, it is cloud-agnostic, and it makes sense to build other open source software on top of it. It also has a dedicated following with over [40,000 contributors](#), and because a lot of developers are already familiar with Kubernetes, it's easier for users to integrate open source solutions built on top of K8s.

Breaking down Kubernetes into building blocks

The simplest way to break down Kubernetes is by looking at the core concepts of container orchestrators. There are containers, which serve as foundational building blocks of work, and then there are the components built on top of each other to tie the system together.

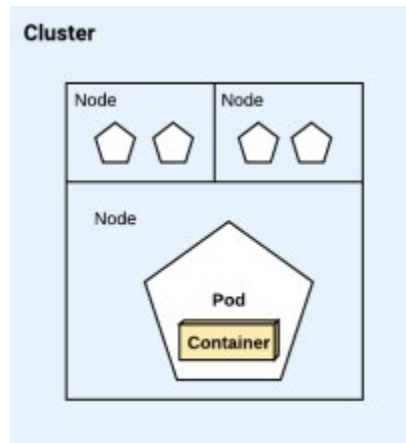
Components come in two core types:

1. **Workload managers:** A way to host and run the containers
2. **Cluster managers:** Global ways to make decisions on behalf of the cluster

In Kubernetes lingo, these roles are fulfilled by the worker nodes and the control plane that manages the work ([Kubernetes components](#)).

Managing the workload

Kubernetes worker nodes have a nested layer of components. At the base layer is the container itself.



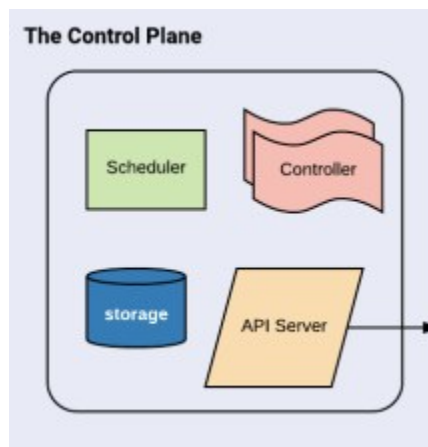
Technically, containers run in pods, which are the atomic object type within a Kubernetes cluster. Here's how they relate:

- **Pod:** A pod defines the logical unit of the application; it can contain one or more containers and each pod is deployed onto a node.
- **Node:** This is the virtual machine serving as the worker in the cluster; pods run on the nodes.
- **Cluster:** This consists of worker nodes and is managed by the control plane.

Each node runs an agent known as the [kubelet](#) for running containers in a pod and a [kube-proxy](#) for managing network rules.

Managing the cluster

The worker nodes manage the containers, and the Kubernetes control plane makes global decisions about the cluster.



The control plane consists of several essential components:

- **Memory store ([etcd](#)):** This is the backend store for all cluster data. While it's possible to run a Kubernetes cluster with a different backing store, etcd, an open source distributed key-value store, is the default.
- **Scheduler ([kube-scheduler](#)):** The scheduler is responsible for assigning newly created pods to the appropriate nodes.
- **API frontend ([kube-apiserver](#)):** This is the gateway from which the developer can interact with Kubernetes—to deploy services, fetch metrics, check logs, etc.
- **Controller manager ([kube-controller-manager](#)):** This watches the cluster and makes necessary changes in order to keep the cluster in the desired state—such as scaling up nodes, maintaining the correct number of pods per replication controller, and creating new namespaces.

The control plane makes decisions to ensure regular operation of the cluster and abstracts away these decisions so that the developer doesn't have to worry about them. Its functionality is highly complex, and users of the system need to have awareness of the logical constraints of the control plane without getting too bogged down on the details.

Using controllers and templates

The components of the cluster dictate how the cluster manages itself—but how do developers or (human) operators tell the cluster how to run the software? This is where [controllers](#) and templates come in.

Controllers orchestrate the pods, and K8s has different types of controllers for different use cases. But the key ones are [Jobs](#), for one-off jobs that run to completion, and [ReplicaSets](#), for running a specified set of identical pods that provide a service.

Like everything else in Kubernetes, these concepts form the building blocks of more complex systems that allow developers to run resilient services. Instead of using ReplicaSets directly, you're encouraged to use [Deployments](#) instead. Deployments manage ReplicaSets on behalf of the user and allow for rolling updates. Kubernetes Deployments ensure that only some pods are down while they're being updated, thereby allowing for zero-downtime deploys. Likewise, [CronJobs](#) manage Jobs and are used for running scheduled and repeated processes. The many layers of K8s allow for better customization, but CronJobs and Deployments suffice for most use cases.

Once you know which controller to pick to run your service, you'll need to configure it with templating.

Anatomy of the template

The Kubernetes template is a YAML file that defines the parameters by which the containers run. Much like any kind of configuration as code, it has its own specific format and requirements that can be a lot to learn. Thankfully, the information you need to provide is the same as if you were running your code against any container orchestrator:

- Tell it what to name the application
- Tell it where to look for the image of the container (often called the container registry)
- Tell it how many instances to run (in the terminology above, the number of ReplicaSets)

```

1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: app-deployment
5   labels:
6     app: app
7 spec:
8   replicas: 3
9   selector:
10    matchLabels:
11      app: app
12      release: production
13 template:
14   metadata:
15     labels:
16       app: app
17       release: production
18   spec:
19     containers:
20     - name: app
21       image: app:1.5.2
22       ports:
23     - containerPort: 80

```

Annotations in the image:

- Type of Resource** points to `kind: Deployment` (line 2).
- Name of the application** points to `name: app-deployment` (line 4).
- Number of Replicas** points to `replicas: 3` (line 8).
- Image Name** points to `image: app:1.5.2` (line 21).

Flexibility in configuration is one of the many advantages of Kubernetes. With the different resources and templates, you can also provide the cluster information about:

- Environment variables
- Location of secrets
- Any data volumes that should be mounted for use by the containers
- How much CPU and memory each container or pod is allowed to use
- The specific command the container should run

And the list goes on.

Bringing it all together

Combining templates from different [resources](#) allows the user to interoperate the components within Kubernetes and customize them for their own needs.

In a bigger ecosystem, developers leverage Jobs, [Services](#), and Deployments with [ConfigMaps](#) and [Secrets](#) that combine to make an application—all of which need to be carefully orchestrated during deployment.

Managing these coordinated steps can be done manually or with one of the common package-management options. While it's definitely possible to roll your own deployment against the Kubernetes API, it's often a good idea to package your configuration—especially if you're shipping open source software that might be deployed and managed by someone not directly on your team.

The package manager of choice for Kubernetes is [Helm](#). It doesn't take a lot to [get started with Helm](#), and it allows you to package your own software for easy installation on a Kubernetes cluster.

Smooth sailing!

The many layers and extensions sitting on top of containers can make container orchestrators difficult to understand. But it's actually all very elegant once you've broken down the pieces and see how they interact. Much like a real orchestra, you develop an appreciation for each individual instrument and watch the harmony come together.

Knowing the fundamentals allows you to recognize and apply patterns and pivot from one container orchestrator to another.

A beginner's guide to load balancing

By Seth Kenlon

When the personal computer was young, a household was likely to have one (or fewer) computers in it. Children played games on it during the day, and parents did accounting or programming or roamed through a BBS in the evening. Imagine a one-computer household today, though, and you can predict the conflict it would create. Everyone would want to use the computer at the same time, and there wouldn't be enough keyboard and mouse to go around.

This is, more or less, the same scenario that's been happening to the IT industry as computers have become more and more ubiquitous. Demand for services and servers has increased to the point that they could grind to a halt from overuse. Fortunately, we now have the concept of load balancing to help us handle the demand.

What is load balancing?

Load balancing is a generic term referring to anything you do to ensure the resources you manage are distributed efficiently. For a web server's systems administrator, load balancing usually means ensuring that the web server software (such as [Nginx](#)) is configured with enough worker nodes to handle a spike in incoming visitors. In other words, should a site suddenly become very popular and its visitor count quadruple in a matter of minutes, the software running the server must be able to respond to each visitor without any of them noticing service degradation. For simple sites, this is as simple as a one-line configuration option, but for complex sites with dynamic content and several database queries for each user, it can be a serious problem.

This problem is supposed to have been solved with cloud computing, but it's not impossible for a web app to fail to scale out when it experiences an unexpected surge.

The important thing to keep in mind when it comes to load balancing is that distributing resources *efficiently* doesn't necessarily mean distributing them *evenly*. Not all tasks require all available resources at all times. A smart load-balancing strategy provides resources to users and tasks only when those resources are needed. This is often the application developer's domain rather than the IT infrastructure's responsibility. Asynchronous applications are vital to ensuring that a user who walks away from the computer for a coffee break isn't occupying valuable resources on the server.

How does load balancing work?

Load balancing avoids bottlenecks by distributing a workload across multiple computational nodes. Those nodes may be physical servers in a data center, containers in a cloud, strategically placed servers enlisted for edge computing, separate Java Virtual Machines (JVMs) in a complex application framework, or daemons running on a single Linux server.

The idea is to divide a large problem into small tasks and assign each task to a dedicated computer. For a website that requires its users to log in, for instance, the website might be hosted on Server A, while the login page and all the authentication lookups that go along with it are hosted on Server B. This way, the process of a new user logging into an account doesn't steal resources from other users actively using the site.

Load balancing the cloud

Cloud computing uses [containers](#), so there aren't usually separate physical servers to handle distinct tasks (actually, there are many separate servers, but they're clustered together to act as one computational "brain"). Instead, a "pod" is created from several containers. When one pod starts to run out of resources due to its user or task load, an identical pod is generated. Pods share storage and network resources, and each pod is assigned to a compute node as it's created. Pods can be created or destroyed on demand as the load requires so that users experience consistent quality of service regardless of how many users there are.

Edge computing

[Edge computing](#) takes the physical world into account when load balancing. The cloud is naturally a distributed system, but in practice, a cloud's nodes are usually concentrated in a few data centers. The further a user is from the data center running the cloud, the more

physical barriers they must overcome for optimal service. Even with fiber connections and proper load balancing, the response time of a server located 3,000 miles away is likely greater than the response time of something just 300 miles away.

Edge computing brings compute nodes to the "edge" of the cloud in an attempt to bridge the geographic divide, forming a sort of satellite network for the cloud, so it also plays a part in a good load-balancing effort.

What is a load-balancing algorithm?

There are many strategies for load balancing, and they range in complexity depending on what technology is involved and what the requirements demand. Load balancing doesn't have to be complicated, and it's important, even when using specialized software like [Kubernetes](#) or [Keepalived](#), to start load balancing from inception.

Don't rely on containers to balance the load when you could design your application to take simple precautions on its own. If you design your application to be modular and ephemeral from the start, then you'll benefit from the load balancing opportunities made available by clever network design, container orchestration, and whatever tomorrow's technology brings.

Some popular algorithms that can guide your efforts as an application developer or network engineer include:

- Assign tasks to servers sequentially (this is often called *round-robin*).
- Assign tasks to the server that's currently the least busy.
- Assign tasks to the server with the best response time.
- Assign tasks randomly.

These principles can be combined or weighted to favor, for instance, the most powerful server in a group when assigning particularly complex tasks. [Orchestration](#) is commonly used so that an administrator doesn't have to drum up the perfect algorithm or strategy for load balancing, although sometimes it's up to the admin to choose which combination of load balancing schemes to use.

Expect the unexpected

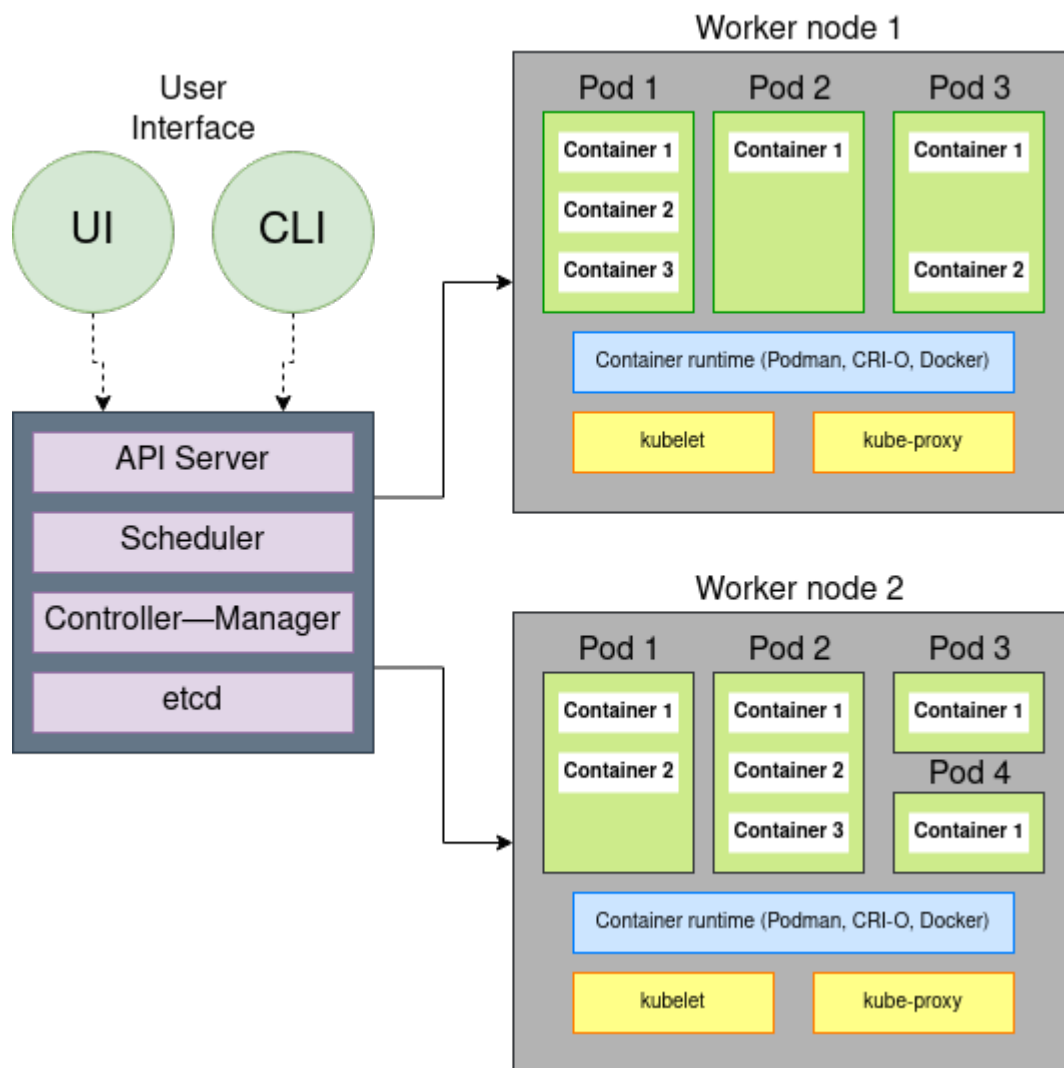
Load balancing isn't really about ensuring that all your resources are used evenly across your network. Load balancing is all about guaranteeing a reliable user experience even when the unexpected happens. Good infrastructure can withstand a computer crash, application

overload, onslaught of network traffic, and user errors. Think about how your service can be resilient and design load balancing accordingly from the ground up.

A guide to Kubernetes architecture

By Nived Velayudhan

You use Kubernetes to orchestrate containers. It's an easy description to say, but understanding what that actually means and how you accomplish it is another matter entirely. If you're running or managing a Kubernetes cluster, then you know that Kubernetes consists of one computer that gets designated as the *control plane*, and lots of other computers that get designated as *worker nodes*. Each of these has a complex but robust stack making orchestration possible, and getting familiar with each component helps understand how it all works.



(Nived Velayudhan, [CC BY-SA 4.0](https://creativecommons.org/licenses/by-sa/4.0/))

Control plane components

You install Kubernetes on a machine called the control plane. It's the one running the Kubernetes daemon, and it's the one you communicate with when starting containers and pods. The following sections describe the control plane components.

Etcd

Etcd is a fast, distributed, and consistent key-value store used as a backing store for persistently storing Kubernetes object data such as pods, replication controllers, secrets, and services. Etcd is the only place where Kubernetes stores cluster state and metadata. The only

component that talks to etcd directly is the Kubernetes API server. All other components read and write data to etcd indirectly through the API server.

Etcd also implements a watch feature, which provides an event-based interface for asynchronously monitoring changes to keys. Once you change a key, its watchers get notified. The API server component heavily relies on this to get notified and move the current state of etcd towards the desired state.

Why should the number of etcd instances be an odd number?

You would typically have three, five, or seven etcd instances running in a high-availability (HA) environment, but why? Because etcd is a distributed data store. It is possible to scale it horizontally but also you need to ensure that the data in each instance is consistent, and for this, your system needs to reach a consensus on what the state is. Etcd uses the [RAFT consensus algorithm](#) for this.

The algorithm requires a majority (or quorum) for the cluster to progress to the next state. If you have only two etcd instances and either of them fails, the etcd cluster can't transition to a new state because no majority exists. If you have three etcd instances, one instance can fail but still have a majority of instances available to reach a quorum.

API server

The API server is the only component in Kubernetes that directly interacts with etcd. All other components in Kubernetes must go through the API server to work with the cluster state, including the clients (kubectl). The API server has the following functions:

- Provides a consistent way of storing objects in etcd.
- Performs validation of those objects so clients can't store improperly configured objects (which could happen if they write directly to the etcd datastore).
- Provides a RESTful API to create, update, modify, or delete a resource.
- Provides [optimistic concurrency locking](#), so other clients never override changes to an object in the event of concurrent updates.
- Performs authentication and authorization of a request that the client sends. It uses the plugins to extract the client's username, user ID, groups the user belongs to, and determine whether the authenticated user can perform the requested action on the requested resource.
- Responsible for [admission control](#) if the request is trying to create, modify, or delete a resource. For example, AlwaysPullImages, DefaultStorageClass, and ResourceQuota.

- Implements a watch mechanism (similar to etcd) for clients to watch for changes. This allows components such as the Scheduler and Controller Manager to interact with the API Server in a loosely coupled manner.

Controller Manager

In Kubernetes, controllers are control loops that watch the state of your cluster, then make or request changes where needed. Each controller tries to move the current cluster state closer to the desired state. The controller tracks at least one Kubernetes resource type, and these objects have a spec field that represents the desired state.

Controller examples:

- Replication Manager (a controller for ReplicationController resources)
- ReplicaSet, DaemonSet, and Job controllers
- Deployment controller
- StatefulSet controller
- Node controller
- Service controller
- Endpoints controller
- Namespace controller
- PersistentVolume controller

Controllers use the watch mechanism to get notified of changes. They watch the API server for changes to resources and perform operations for each change, whether it's a creation of a new object or an update or deletion of an existing object. Most of the time, these operations include creating other resources or updating the watched resources themselves. Still, because using watches doesn't guarantee the controller won't miss an event, they also perform a re-list operation periodically to ensure they haven't missed anything.

The Controller Manager also performs lifecycle functions such as namespace creation and lifecycle, event garbage collection, terminated-pod garbage collection, [cascading-deletion garbage collection](#), and node garbage collection. See [Cloud Controller Manager](#) for more information.

Scheduler

The Scheduler is a control plane process that assigns pods to nodes. It watches for newly created pods that have no nodes assigned. For every pod that the Scheduler discovers, the Scheduler becomes responsible for finding the best node for that pod to run on.

Nodes that meet the scheduling requirements for a pod get called feasible nodes. If none of the nodes are suitable, the pod remains unscheduled until the Scheduler can place it. Once it finds a feasible node, it runs a set of functions to score the nodes, and the node with the highest score gets selected. It then notifies the API server about the selected node. They call this process binding.

The selection of nodes is a two-step process:

1. Filtering the list of all nodes to obtain a list of acceptable nodes to which you can schedule the pod (for example, the PodFitsResources filter checks whether a candidate node has enough available resources to meet a pod's specific resource requests).
2. Scoring the list of nodes obtained from the first step and ranking them to choose the best node. If multiple nodes have the highest score, a round-robin process ensures the pods get deployed across all of them evenly.

Factors to consider for scheduling decisions include:

- Does the pod request hardware/software resources? Is the node reporting a memory or a disk pressure condition?
- Does the node have a label that matches the node selector in the pod specification?
- If the pod requests binding to a specific host port, is that port available?
- Does the pod tolerate the taints of the node?
- Does the pod specify node affinity or anti-affinity rules?

The Scheduler doesn't instruct the selected node to run the pod. All the Scheduler does is update the pod definition through the API server. The API server then notifies the kubelet that the pod got scheduled through the watch mechanism. Then the kubelet service on the target node sees that the pod got scheduled to its node, it creates and runs the pod's containers.

Worker node components

Worker nodes run the kubelet agent, which permits them to get recruited by the control plane to process jobs. Similar to the control plane, the worker node uses several different components to make this possible. The following sections describe the worker node components.

Kubelet

Kubelet is an agent that runs on each node in the cluster and is responsible for everything running on a worker node. It ensures that the containers run in the pod.

The main functions of kubelet service are:

- Register the node it's running on by creating a node resource in the API server.
- Continuously monitor the API server for pods that got scheduled to the node.
- Start the pod's containers by using the configured container runtime.
- Continuously monitor running containers and report their status, events, and resource consumption to the API server.
- Run the container liveness probes, restart containers when the probes fail and terminate containers when their pod gets deleted from the API server (notifying the server about the pod termination).

Service proxy

The service proxy (kube-proxy) runs on each node and ensures that one pod can talk to another pod, one node can talk to another node, and one container can talk to another container. It is responsible for watching the API server for changes on services and pod definitions to maintain that the entire network configuration is up to date. When a service gets backed by more than one pod, the proxy performs load balancing across those pods.

The kube-proxy got its name because it began as an actual proxy server that used to accept connections and proxy them to the pods. The current implementation uses iptables rules to redirect packets to a randomly selected backend pod without passing them through an actual proxy server.

A high-level view of how it works:

- When you create a service, a virtual IP address gets assigned immediately.
- The API server notifies the kube-proxy agents running on worker nodes that a new service exists.
- Each kube-proxy makes the service addressable by setting up iptables rules, ensuring each service IP/port pair gets intercepted and the destination address gets modified to one of the pods that back the service.
- Watches the API server for changes to services or its endpoint objects.

Container runtime

There are two categories of container runtimes:

- **Lower-level container runtimes:** These focus on running containers and setting up the namespace and cgroups for containers.
- **Higher-level container runtimes (container engine):** These focus on formats, unpacking, management, sharing of images, and providing APIs for developers.

Container runtime takes care of:

- Pulls the required container image from an image registry if it's not available locally.
- Extracts the image onto a copy-on-write filesystem and all the container layers overlay to create a merged filesystem.
- Prepares a container mount point.
- Sets the metadata from the container image like overriding CMD, ENTRYPOINT from user inputs, and sets up SECCOMP rules, ensuring the container runs as expected.
- Alters the kernel to assign isolation like process, networking, and filesystem to this container.
- Alerts the kernel to assign some resource limits like CPU or memory limits.
- Pass system call (syscall) to the kernel to start the container.
- Ensures that the SELinux/AppArmor setup is proper.

Working together

System-level components work together to ensure that each part of a Kubernetes cluster can realize its purpose and perform its functions. It can sometimes be overwhelming (when you're deep into editing a [YAML file](#)) to understand how your requests get communicated within your cluster. Now that you have a map of how the pieces fit together, you can better understand what's happening inside Kubernetes, which helps you diagnose problems, maintain a healthy cluster, and optimize your own workflow.

Getting started with OKD on your Linux desktop

By: Ricardo Gerardi

OKD is the open source upstream community edition of Red Hat's OpenShift container platform. OKD is a container management and orchestration platform based on OCI containers and Kubernetes.

OKD is a complete solution to manage, deploy, and operate containerized applications that includes an easy-to-use web interface, automated build tools, routing capabilities, and monitoring and logging aggregation features. Of course, it also has all the features provided by Kubernetes natively.

OKD provides several deployment options aimed at different requirements with single or multiple master nodes, high-availability capabilities, logging, monitoring, and more. You can create OKD clusters as small or as large as you need.

In addition to these deployment options, OKD provides a way to create a local, all-in-one cluster on your own machine using the **oc** command-line tool. This is a great option if you want to try OKD locally without committing the resources to create a larger multi-node cluster, or if you want to have a local cluster on your machine as part of your workflow or development process. In this case, you can create and deploy the applications locally using the same APIs and interfaces required to deploy the application on a larger scale. This process ensures a seamless integration that prevents issues with applications that work in the developer's environment but not in production.

This tutorial shows you how to create an OKD cluster using **oc cluster up** in a Linux box.

1. Install Docker

The **oc cluster up** command creates a local OKD cluster on your machine using Docker containers. In order to use this command, you need Podman or Docker installed on your machine. If either Podman or Docker is not installed on your system, install one by using your distribution's package manager. For example, on Fedora, CentOS, or RHEL:

```
$ sudo dnf install podman
```

Or use Docker:

```
$ sudo dnf install docker
```

2. Configure Docker insecure registry

When using Docker, you need to configure it to allow the communication with an insecure registry on address 172.30.0.0/16. This insecure registry will be deployed with your local OKD cluster later.

On CentOS or RHEL, edit the file **/etc/docker/daemon.json** by adding these lines:

```
{  
  "insecure-registries": ["172.30.0.0/16"]  
}
```

On Fedora, edit the file **/etc/containers/registries.conf** by adding these lines:

```
[registries.insecure]  
registries = ['172.30.0.0/16']
```

3. Start Docker

Docker operates through a daemon, so you must start Docker, create a system group named **docker** and assign this group to your user so you can run Docker commands with your own user, without requiring root or sudo access. This allows you to create your OKD cluster using your own user. If you're using Podman, this step isn't required because Podman doesn't use a daemon.

For example, these are the commands to create the group and assign it to my local user, **ricardo**:

```
$ sudo groupadd docker
$ sudo usermod -a -G docker ricardo
```

You need to log out and log back in to see the new group association. After logging back in, run the **id** command and ensure you're a member of the **docker** group:

```
$ id
uid=1000(ricardo) gid=1000(ricardo) groups=1000(ricardo),10(wheel),1001(docker)
context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

Now start and enable the Docker daemon like this:

```
$ sudo systemctl start docker
$ sudo systemctl enable docker
Created symlink from /etc/systemd/system/multi-user.target.wants/docker.service
to /usr/lib/systemd/system/docker.service.
```

Verify that Docker is running:

```
$ docker version
Client:
Version:      1.13.1
API version:  1.26
Package version: docker-1.13.1-75.git8633870.el7.centos.x86_64
Go version:   go1.9.4
Git commit:   8633870/1.13.1
Built:        Fri Sep 28 19:45:08 2018
OS/Arch:      linux/amd64
[...]
```

Ensure that the insecure registry option has been enabled by running **docker info** and looking for these lines:

```
$ docker info
... Skipping long output ...
Insecure Registries:
 172.30.0.0/16
 127.0.0.0/8
```

4. Open firewall ports

Whether you're using Podman or Docker, you must next open firewall ports to ensure your OKD containers can communicate with the master API. By default, some distributions have the firewall enabled, which blocks required connectivity from the OKD containers to the

master API. If your system has the firewall enabled, you need to add rules to allow communication on ports **8443/tcp** for the master API and **53/udp** for DNS resolution on the Docker bridge subnet.

For CentOS, RHEL, and Fedora, you can use the **firewall-cmd** command-line tool to add the rules. For other distributions, you can use the provided firewall manager, such as [UFW](#) or [iptables](#).

Before adding the firewall rules, obtain the Docker bridge network subnet's address, like this:

```
$ docker network inspect bridge | grep Subnet
"Subnet": "172.17.0.0/16",
```

Enable the firewall rules using this subnet. For CentOS, RHEL, and Fedora, use **firewall-cmd** to add a new zone:

```
$ sudo firewall-cmd --permanent --new-zone okdlocal
success
```

Include the subnet address you obtained before as a source to the new zone:

```
$ sudo firewall-cmd --permanent --zone okdlocal --add-source 172.17.0.0/16
success
```

Next, add the required rules to the **okdlocal** zone:

```
$ sudo firewall-cmd --permanent --zone okdlocal --add-port 8443/tcp
success
$ sudo firewall-cmd --permanent --zone okdlocal --add-port 53/udp
success
$ sudo firewall-cmd --permanent --zone okdlocal --add-port 8053/udp
success
```

Finally, reload the firewall to enable the new rules:

```
$ sudo firewall-cmd --reload
success
```

Ensure that the new zone and rules are in place:

```
$ sudo firewall-cmd --zone okdlocal --list-sources
172.17.0.0/16
$ sudo firewall-cmd --zone okdlocal --list-ports
8443/tcp 53/udp 8053/udp
```

Your system is ready to start the cluster. It's time to download the OKD client tools.

5. Download the OKD client tools

To deploy a local OKD cluster using **oc**, you need to download the OKD client tools package. For some distributions, like CentOS and Fedora, this package can be downloaded as an RPM from the official repositories. Please note that these packages may follow the distribution update cycle and usually are not the most recent version available.

For this tutorial, download the OKD client package directly from the official GitHub repository so you can get the most recent version available. At the time of writing, this was OKD v3.11.

Go to the [OKD downloads page](#) to get the link to the OKD tools for Linux, then download it with **wget**:

```
$ cd ~/Downloads/
$ wget https://github.com/openshift/origin/releases/download/v3.11.0/openshift-
origin-client-tools-v3.11.0-0cbc58b-linux-64bit.tar.gz
```

Uncompress the downloaded package:

```
$ tar -xzvf openshift-origin-client-tools-v3.11.0-0cbc58b-linux-64bit.tar.gz
```

Finally, to make it easier to use the **oc** command systemwide, move it to a directory included in your **\$PATH** variable. A good location is **/usr/local/bin**:

```
$ sudo cp openshift-origin-client-tools-v3.11.0-0cbc58b-linux-64bit/oc
/usr/local/bin/
```

One of the nicest features of the **oc** command is that it's a static single binary. You don't need to install it to use it.

Check that the **oc** command is working:

```
$ oc version
oc v3.11.0+0cbc58b
kubernetes v1.11.0+d4cacc0
features: Basic-Auth GSSAPI Kerberos SPNEGO
```

6. Start your OKD cluster

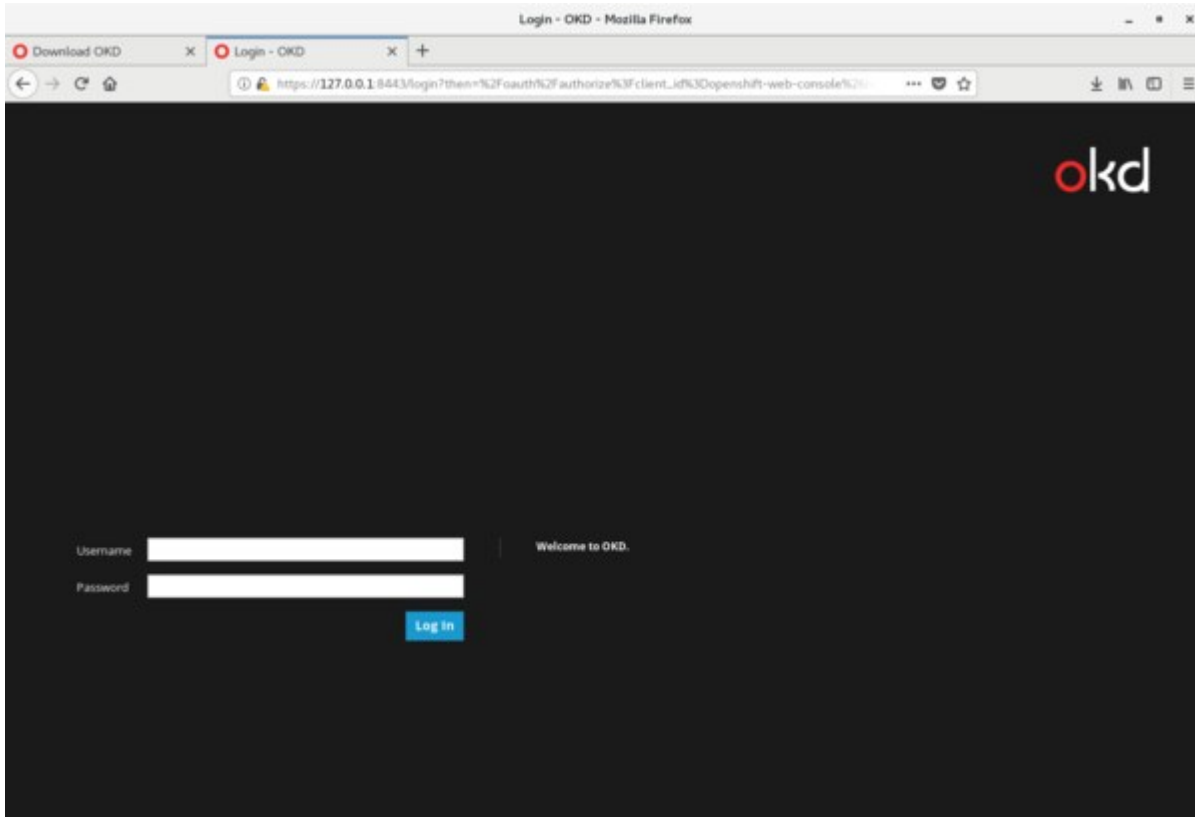
Once you have all the prerequisites in place, start your local OKD cluster by running this command:

```
$ oc cluster up
```

This command downloads all required images from Docker Hub, and starts the containers. The first time you run it, it takes a few minutes to complete. When it's finished, you see:

```
... Skipping long output ...
OpenShift server started.
The server is accessible via web console at:
  https://127.0.0.1:8443
You are logged in as:
  User:      developer
  Password: <any value>
To login as administrator:
  oc login -u system:admin
```

Access the OKD web console by using the browser and navigating to <https://127.0.0.1:8443>



From the command line, you can check if the cluster is running by entering this command:

```
$ oc cluster status
Web console URL: https://127.0.0.1:8443/console/
Config is at host directory
Volumes are at host directory
Persistent volumes are at host directory
/home/ricardo/openshift.local.clusterup/openshift.local.pv
Data will be discarded when cluster is destroyed
```

You can also verify your cluster is working by logging in as the **system:admin** user and checking available nodes using the **oc** command-line tool:

```
$ oc login -u system:admin
Logged into "https://127.0.0.1:8443" as "system:admin" using existing
credentials.
You have access to the following projects and can switch between them with 'oc
project <projectname>':
  default
  kube-dns
  kube-proxy
  kube-public
```

```

    kube-system
  * myproject
    openshift
    openshift-apiserver
    openshift-controller-manager
    openshift-core-operators
    openshift-infra
    openshift-node
    openshift-service-cert-signer
    openshift-web-console
Using project "myproject".
$ oc get nodes
NAME             STATUS    ROLES    AGE      VERSION
localhost        Ready    <none>   52m      v1.11.0+d4cacc0

```

Since this is a local, all-in-one cluster, you see only **localhost** in the nodes list.

7. Smoke-test your cluster

Now that your local OKD cluster is running, create a test app to smoke-test it. Use OKD to build and start the sample application so you can ensure the different components are working.

Start by logging in as the **developer** user:

```

$ oc login -u developer
Logged into "https://127.0.0.1:8443" as "developer" using existing credentials.
You have one project on this server: "myproject"
Using project "myproject".

```

You're automatically assigned to a new, empty project named **myproject**. Create a sample PHP application based on an existing GitHub repository, like this:

```

$ oc new-app php:5.6~https://github.com/rgerardi/ocp-smoke-test.git
--> Found image 92ed8b3 (5 months old) in image stream "openshift/php" under tag
"5.6" for "php:5.6"
    Apache 2.4 with PHP 5.6
    -----
    PHP 5.6 available as container is a base platform for building and running
various PHP 5.6 applications and frameworks. PHP is an HTML-embedded scripting
language. [...]
    Tags: builder, php, php56, rh-php56
    * A source build using source code from https://github.com/rgerardi/ocp-
smoke-test.git will be created

```

```

    * The resulting image will be pushed to image stream tag "ocp-smoke-
test:latest"
    * Use 'start-build' to trigger a new build
    * This image will be deployed in deployment config "ocp-smoke-test"
    * Ports 8080/tcp, 8443/tcp will be load balanced by service "ocp-smoke-test"
    * Other containers can access this service through the hostname "ocp-smoke-
test"
--> Creating resources ...
    imagestream.image.openshift.io "ocp-smoke-test" created
    buildconfig.build.openshift.io "ocp-smoke-test" created
    deploymentconfig.apps.openshift.io "ocp-smoke-test" created
    service "ocp-smoke-test" created
--> Success
    Build scheduled, use 'oc logs -f bc/ocp-smoke-test' to track its progress.
    Application is not exposed. You can expose services to the outside world by
xecuting one or more of the commands below:
    'oc expose svc/ocp-smoke-test'
    Run 'oc status' to view your app.

```

OKD starts the build process, which clones the provided GitHub repository, compiles the application (if required), and creates the necessary images. You can follow the build process by tailing its log with this command:

```

$ oc logs -f bc/ocp-smoke-test
Cloning "https://github.com/rgerardi/ocp-smoke-test.git" ...
    Commit: 391a475713d01ab0afab700bab8a3d7549c5cc27 (Create index.php)
    Author: Ricardo Gerardi <ricardo.gerardi@gmail.com>
    Date:   Tue Oct 2 13:47:25 2018 -0400

Using
172.30.1.1:5000/openshift/php@sha256:f3c95020fa870fcefa7d1440d07a2b947834b87bdaf0
00588e84ef4a599c7546 as the s2i builder image
---> Installing application source...
=> sourcing 20-copy-config.sh ...
---> 04:53:28      Processing additional arbitrary httpd configuration provided by
s2i ...
=> sourcing 00-documentroot.conf ...
=> sourcing 50-mpm-tuning.conf ...
=> sourcing 40-ssl-certs.sh ...
Pushing image 172.30.1.1:5000/myproject/ocp-smoke-test:latest ...
Pushed 1/10 layers, 10% complete
Push successful

```

After the build process completes, OKD starts the application automatically by running a new pod based on the created image. You can see this new pod with this command:

```

$ oc get pods

```

NAME	READY	STATUS	RESTARTS	AGE
ocp-smoke-test-1-build	0/1	Completed	0	1m
ocp-smoke-test-1-d8h76	1/1	Running	0	7s

You can see two pods are created; the first one (with the status Completed) is the pod used to build the application. The second one (with the status Running) is the application itself.

In addition, OKD creates a service for this application. Verify it by using this command:

```
$ oc get service
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
ocp-smoke-test	ClusterIP	172.30.232.241	<none>	8080/TCP, 8443/TCP

Finally, expose this service externally using OKD routes so you can access the application from a local browser:

```
$ oc expose svc ocp-smoke-test
route.route.openshift.io/ocp-smoke-test exposed
$ oc get route
```

NAME	HOST/PORT	PATH	SERVICES	PORT	TERMINATION	WILDCARD
ocp-smo..	ocp-smoke-..	ct.127.0.0.1.nip.io		ocp-smoke-test	8080-tcp	None

Verify that your new application is running by navigating to <http://ocp-smoke-test-myproject.127.0.0.1.nip.io> in a web browser:

phpinfo() - Mozilla Firefox

Download OKD x OpenShift Web Console x phpinfo() x +

ocp-smoke-test-myproject.127.0.0.1.nip.io

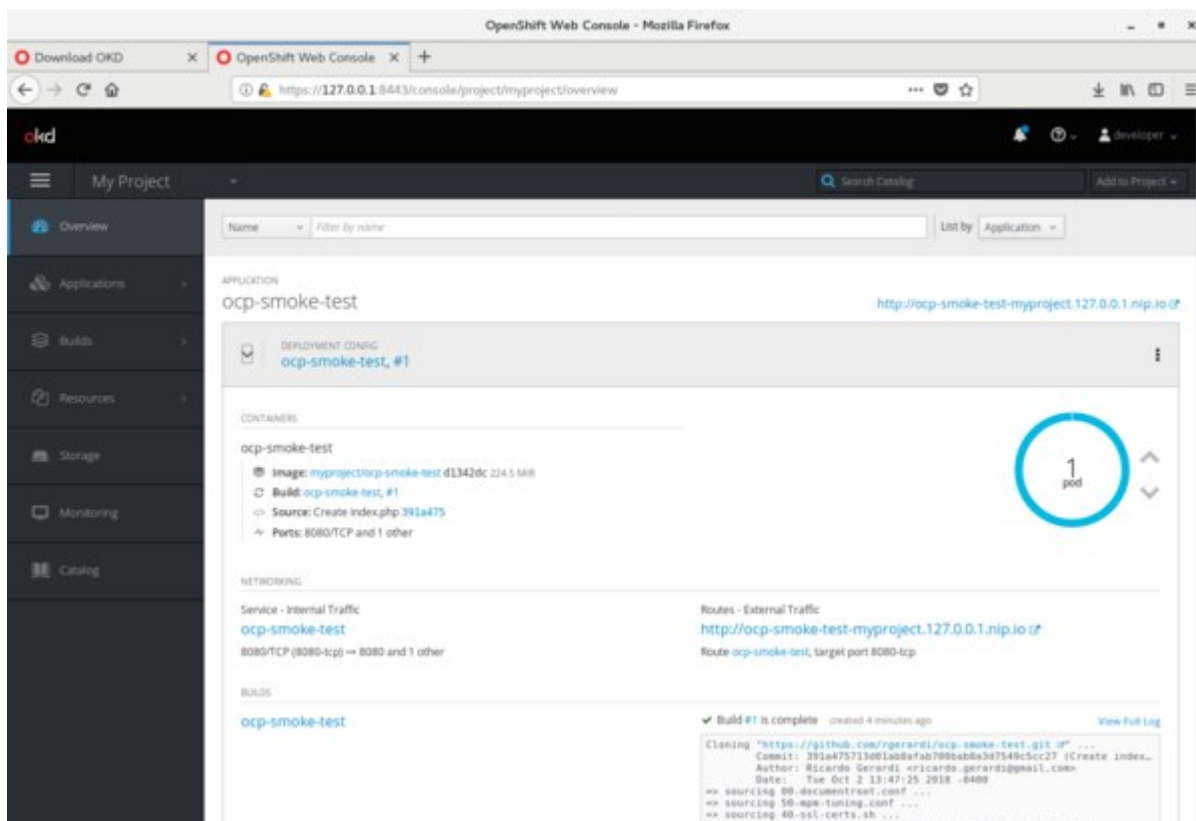
PHP Version 5.6.25



System	Linux ocp-smoke-test-1-d8h76 3.10.0-862.14.4.el7.x86_64 #1 SMP Wed Sep 26 15:12:11 UTC 2018 x86_64
Build Date	Oct 21 2016 18:01:06
Server API	Apache 2.0 Handler
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc/opt/rh/rh-php56
Loaded Configuration File	/etc/opt/rh/rh-php56/php.ini
Scan this dir for additional .ini files	/etc/opt/rh/rh-php56/php.d
Additional .ini files parsed	/etc/opt/rh/rh-php56/php.d/10-opcache.ini, /etc/opt/rh/rh-php56/php.d/15-xdebug.ini, /etc/opt/rh/rh-php56/php.d/20-bcmath.ini, /etc/opt/rh/rh-php56/php.d/20-bz2.ini, /etc/opt/rh/rh-php56/php.d/20-calendar.ini, /etc/opt/rh/rh-php56/php.d/20-ctype.ini, /etc/opt/rh/rh-php56/php.d/20-curl.ini, /etc/opt/rh/rh-php56/php.d/20-dom.ini, /etc/opt/rh/rh-php56/php.d/20-exif.ini, /etc/opt/rh/rh-php56/php.d/20-fileinfo.ini, /etc/opt/rh/rh-php56/php.d/20-ftp.ini, /etc/opt/rh/rh-php56/php.d/20-gd.ini, /etc/opt/rh/rh-php56/php.d/20-gettext.ini, /etc/opt/rh/rh-php56/php.d/20-gmp.ini, /etc/opt/rh/rh-php56/php.d/20-iconv.ini, /etc/opt/rh/rh-php56/php.d/20-intl.ini, /etc/opt/rh/rh-php56/php.d/20-ldap.ini, /etc/opt/rh/rh-php56/php.d/20-mbstring.ini, /etc/opt/rh/rh-php56/php.d/20-mysqli.ini, /etc/opt/rh/rh-php56/php.d/20-pdo.ini, /etc/opt/rh/rh-php56/php.d/20-pgsql.ini, /etc/opt/rh/rh-php56/php.d/20-phar.ini, /etc/opt/rh/rh-php56/php.d/20-posix.ini, /etc/opt/rh/rh-php56/php.d/20-sockets.ini, /etc/opt/rh/rh-php56/php.d/20-soap.ini, /etc/opt/rh/rh-php56/php.d/20-sockets.ini, /etc/opt/rh/rh-php56/php.d/20-sqlite3.ini, /etc/opt/rh/rh-php56/php.d/20-sysmsg.ini, /etc/opt/rh/rh-php56/php.d/20-syssem.ini, /etc/opt/rh/rh-php56/php.d/20-sysvshm.ini, /etc/opt/rh/rh-php56/php.d/20-tokenizer.ini, /etc/opt/rh/rh-php56/php.d/20-xml.ini, /etc/opt/rh/rh-php56/php.d/20-xmlreader.ini, /etc/opt/rh/rh-php56/php.d/20-xmlwriter.ini, /etc/opt/rh/rh-php56/php.d/20-xsl.ini, /etc/opt/rh/rh-php56/php.d/20-zip.ini, /etc/opt/rh/rh-php56/php.d/30-mysql.ini, /etc/opt/rh/rh-php56/php.d/30-mysqli.ini, /etc/opt/rh/rh-php56/php.d/30-pdo_mysql.ini, /etc/opt/rh/rh-php56/php.d/30-pdo_pgsql.ini, /etc/opt/rh/rh-php56/php.d/30-pdo_sqlite.ini, /etc/opt/rh/rh-php56/php.d/30-pdo_sqlite.ini, /etc/opt/rh/rh-php56/php.d/30-wddx.ini, /etc/opt/rh/rh-php56/php.d/30-xmlreader.ini, /etc/opt/rh/rh-php56/php.d/40-json.ini, /etc/opt/rh/rh-php56/php.d/40-memcache.ini
PHP API	20131106
PHP Extension	20131226
Zend Extension	220131226
Zend Extension Build	API20131226.NTS
PHP Extension Build	API20131226.NTS
Debug Build	no
Thread Safety	disabled
Zend Signal Handling	disabled
Zend Memory Manager	enabled
Zend Multibyte Support	provided by mbstring

OKD sample web application

You can also see the status of your application by logging into the OKD web console:



Learn more

You can find more information about OKD on the [official site](#), which includes a link to the [OKD documentation](#).

If this is your first time working with OKD/OpenShift, you can learn the basics of the platform, including how to build and deploy containerized applications, through the [Interactive Learning Portal](#). Another good resource is the official [OpenShift YouTube channel](#).