# Kubernetes: an open-source container orchestration platform

Dott. Costantino Zazza, costantino.zazza@unitus.it

**IR0000032 – ITINERIS, Italian Integrated Environmental Research Infrastructures System**
(D.D. n. 130/2022 - CUP B53C22002150006) Funded by EU - Next Generation EU PNRR-Mission 4 “Education and Research” - Component 2: “From research to business” - Investment 3.1: “Fund for the realisation of an integrated system of research and innovation infrastructures”

Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani

## About Docker

Docker is the container packaging and runtime standard. Developers build container images from Dockerfiles and distribute container images from Docker registries. Docker Hub is the most popular public registry. Many organizations also set up private Docker registries. Docker is primarily used to manage containers on individual nodes.

NOTE

*Although Rancher 1.6 supported Docker Swarm clustering technology, it is no longer supported in Rancher 2.x due to the success of Kubernetes.*

## About Kubernetes

Kubernetes is the container cluster management standard. YAML files specify containers and other resources that form an application. Kubernetes performs functions such as scheduling, scaling, service discovery, health check, secret management, and configuration management.

## What is a Kubernetes Cluster?

A cluster is a group of computers that work together as a single system.

A *Kubernetes Cluster* is a cluster that uses the Kubernetes container-orchestration system to deploy, maintain, and scale Docker containers, allowing your organization to automate application operations.

## Roles for Nodes in Kubernetes Clusters

Each computing resource in a Kubernetes cluster is called a *node*. Nodes can be either bare-metal servers or virtual machines. Kubernetes classifies nodes into three types: *etcd* nodes, *control plane* nodes, and *worker* nodes.

*A Kubernetes cluster consists of at least one **etcd**, **controlplane**, and **worker** node.*

## etcd Nodes

Rancher uses etcd as a data store in both single node and high-availability installations. In Kubernetes, etcd is also a role for nodes that store the cluster state.

The state of a Kubernetes cluster is maintained in etcd. The etcd nodes run the etcd database.

The etcd database component is a distributed key-value store used as Kubernetes storage for all cluster data, such as cluster coordination and state management. It is recommended to run etcd on multiple nodes so that there's always a backup available for failover.

Although you can run etcd on just one node, etcd requires a majority of nodes, a quorum, to agree on updates to the cluster state. The cluster should always contain enough healthy etcd nodes to form a quorum. For a cluster with n members, a quorum is (n/2)+1. For any odd-sized cluster, adding one node will always increase the number of nodes necessary for a quorum.

***Three*** etcd nodes is generally sufficient for smaller clusters and ***five*** etcd nodes for large clusters.

## Controlplane Nodes

Controlplane nodes run the Kubernetes API server, scheduler, and controller manager. These nodes take care of routine tasks to ensure that your cluster maintains your configuration. Because all cluster data is stored on your etcd nodes, control plane nodes are stateless. You can run control plane on a single node, although three or more nodes are recommended for redundancy. Additionally, a single node can share the control plane and etcd roles.

## Worker Nodes

Each worker node runs the following:

**Kubelets:** An agent that monitors the state of the node, ensuring your containers are healthy.

**Workloads:** The containers and pods that hold your apps, as well as other types of deployments.

Worker nodes also run storage and networking drivers, and ingress controllers when required. You create as many worker nodes as necessary to run your workloads.

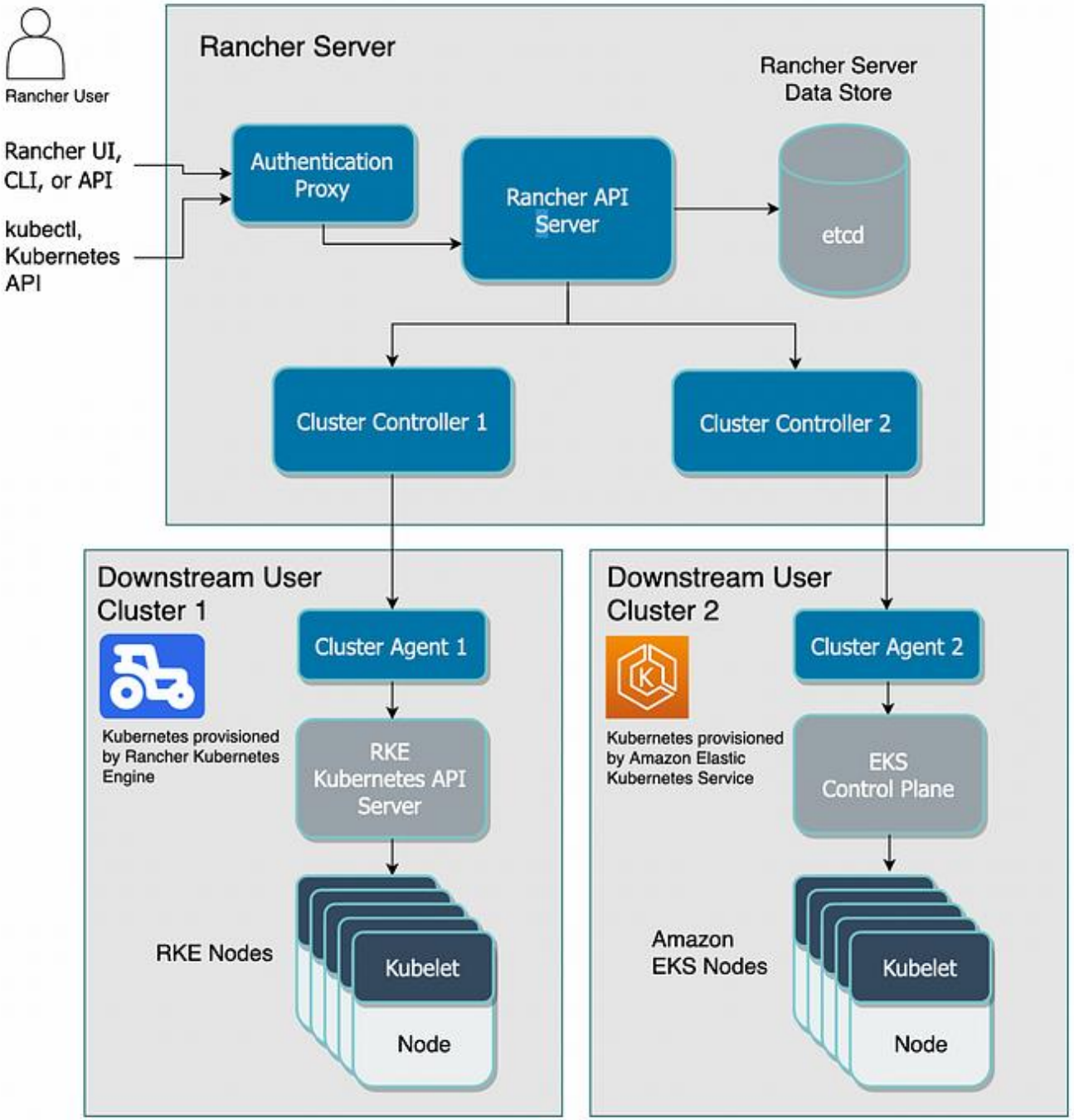
## About Helm

For high-availability installations of Rancher, Helm is the tool used to install Rancher on a Kubernetes cluster.

Helm is the package management tool of choice for Kubernetes. Helm charts provide templating syntax for Kubernetes YAML manifest documents. With Helm we can create configurable deployments instead of just using static files. For more information about creating your own catalog of deployments, check out the docs at https://helm.sh/.

For more information on service accounts and cluster role binding, refer to the Kubernetes documentation.


RANCHER 2.0
COMPLETE CONTAINER MANAGEMENT PLATFORM
Workload Management
User Interface • App Catalog • CI/CD • Monitoring • Logging
Unified Cluster Management
Provisioning • Authentication • RBAC • Policy • Security • Capacity • Cost
Rancher Kubernetes Engine (RKE)
vSphere • Bare Metal
aws
EKS
Google Cloud Platform
GKE
Microsoft Azure
AKS


Rancher User
Rancher Server
Rancher Server Data Store
Rancher UI, CLI, or API
kubectl, Kubernetes API
Authentication Proxy
Rancher API Server
etcd
Cluster Controller 1
Cluster Controller 2
Downstream User Cluster 1
Kubernetes provisioned by Rancher Kubernetes Engine
Cluster Agent 1
RKE Kubernetes API Server
RKE Nodes
Kubelet
Node
Downstream User Cluster 2
Kubernetes provisioned by Amazon Elastic Kubernetes Service
Cluster Agent 2
EKS Control Plane
Amazon EKS Nodes
Kubelet
Node

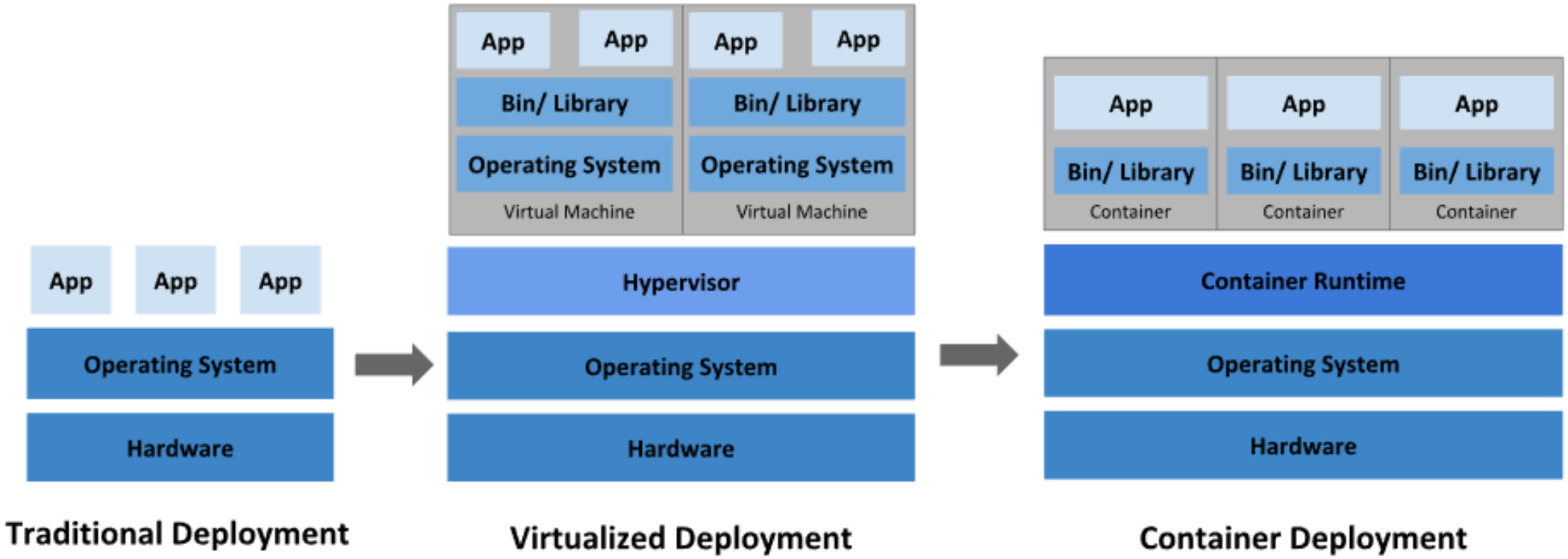
## Introduction to Kubernetes

Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

The name Kubernetes originates from Greek, meaning helmsman or pilot. K8s as an abbreviation results from counting the eight letters between the "K" and the "s". Google open-sourced the Kubernetes project in 2014. Kubernetes combines over 15 years of Google's experience running production workloads at scale with best-of-breed ideas and practices from the community.

## Going back in time

Let's take a look at why Kubernetes is so useful by going back in time...


App
App
App
Operating System
Hardware
Traditional Deployment
App
App
App
App
Bin/ Library
Bin/ Library
Operating System
Operating System
Virtual Machine
Virtual Machine
Hypervisor
Operating System
Hardware
Virtualized Deployment
App
App
App
Bin/ Library
Bin/ Library
Bin/ Library
Container
Container
Container
Container Runtime
Operating System
Hardware
Container Deployment

**Traditional deployment era:** Early on, organizations ran applications on physical servers. There was no way to define resource boundaries for applications in a physical server, and this caused resource allocation issues. For example, if multiple applications run on a physical server, there can be instances where one application would take up most of the resources, and as a result, the other applications would underperform. A solution for this would be to run each application on a different physical server. But this did not scale as resources were underutilized, and it was expensive for organizations to maintain many physical servers.

**Virtualized deployment era:** As a solution, virtualization was introduced. It allows you to run multiple Virtual Machines (VMs) on a single physical server's CPU. Virtualization allows applications to be isolated between VMs and provides a level of security as the information of one application cannot be freely accessed by another application.

Virtualization allows better utilization of resources in a physical server and allows better scalability because an application can be added or updated easily, reduces hardware costs, and much more. With virtualization you can present a set of physical resources as a cluster of disposable virtual machines.

Each VM is a full machine running all the components, including its own operating system, on top of the virtualized hardware.

**Container deployment era:** Containers are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications. Therefore, containers are considered lightweight. Similar to a VM, a container has its own filesystem, share of CPU, memory, process space, and more. As they are decoupled from the underlying infrastructure, they are portable across clouds and OS distributions.

Containers have become popular because they provide extra benefits, such as:

**Agile application** creation and deployment: increased ease and efficiency of container image creation compared to VM image use.

**Continuous development**, integration, and deployment: provides for reliable and frequent container image build and deployment with quick and efficient rollbacks (due to image immutability).

**Dev and Ops** separation of concerns: create application container images at build/release time rather than deployment time, thereby decoupling applications from infrastructure.

**Observability**: not only surfaces OS-level information and metrics, but also application health and other signals.

**Environmental consistency** across development, testing, and production: runs the same on a laptop as it does in the cloud.

**Cloud and OS distribution portability**: runs on Ubuntu, RHEL, CoreOS, on-premises, on major public clouds, and anywhere else.

**Application-centric management:** raises the level of abstraction from running an OS on virtual hardware to running an application on an OS using logical resources.

**Loosely coupled**, distributed, elastic, liberated micro-services: applications are broken into smaller, independent pieces and can be deployed and managed dynamically – not a monolithic stack running on one big single-purpose machine.

**Resource isolation**: predictable application performance.

**Resource utilization**: high efficiency and density.

## Why you need Kubernetes and what it can do

Containers are a good way to bundle and run your applications. In a production environment, you need to manage the containers that run the applications and ensure that there is no downtime. For example, if a container goes down, another container needs to start. Wouldn't it be easier if this behavior was handled by a system?

That's how Kubernetes comes to the rescue! Kubernetes provides you with a framework to run distributed systems resiliently. It takes care of scaling and failover for your application, provides deployment patterns, and more. For example: Kubernetes can easily manage a canary deployment for your system.

Kubernetes provides you with:

**Service discovery and load balancing** Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.

**Storage orchestration** Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.

**Automated rollouts and rollbacks** You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.

**Automatic bin packing** You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.

**Self-healing** Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.

**Secret and configuration management** Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.

**What Kubernetes is not**

**Kubernetes is not a traditional, all-inclusive PaaS (Platform as a Service) system**. Since Kubernetes operates at the container level rather than at the hardware level, it provides some generally applicable features common to PaaS offerings, such as deployment, scaling, load balancing, and lets users integrate their logging, monitoring, and alerting solutions. However, Kubernetes is not monolithic, and these default solutions are optional and pluggable. Kubernetes provides the building blocks for building developer platforms, but preserves user choice and flexibility where it is important.

Kubernetes:

- **Does not limit the types of applications supported**. Kubernetes aims to support an extremely diverse variety of workloads, including stateless, stateful, and data-processing workloads. If an application can run in a container, it should run great on Kubernetes.

- **Does not deploy source code and does not build your application**. Continuous Integration, Delivery, and Deployment (CI/CD) workflows are determined by organization cultures and preferences as well as technical requirements.

- **Does not provide application-level services**, such as middleware (for example, message buses), data-processing frameworks (for example, Spark), databases (for example, MySQL), caches, nor cluster storage systems (for example, Ceph) as built-in services. Such components can run on Kubernetes, and/or can be accessed by applications running on Kubernetes through portable mechanisms, such as the Open Service Broker.

- **Does not dictate logging, monitoring, or alerting solutions**. It provides some integrations as proof of concept, and mechanisms to collect and export metrics.

- **Does not provide nor mandate a configuration language/system** (for example, Jsonnet). It provides a declarative API that may be targeted by arbitrary forms of declarative specifications.

- **Does not provide nor adopt any comprehensive machine configuration**, maintenance, management, or self-healing systems.

*Additionally, Kubernetes is not a mere orchestration system. In fact, it eliminates the need for orchestration. The technical definition of orchestration is execution of a defined workflow: first do A, then B, then C. In contrast, Kubernetes comprises a set of independent, composable control processes that continuously drive the current state towards the provided desired state. It shouldn't matter how you get from A to C. Centralized control is also not required.*

*This results in a system that is easier to use and more powerful, robust, resilient, and extensible.*

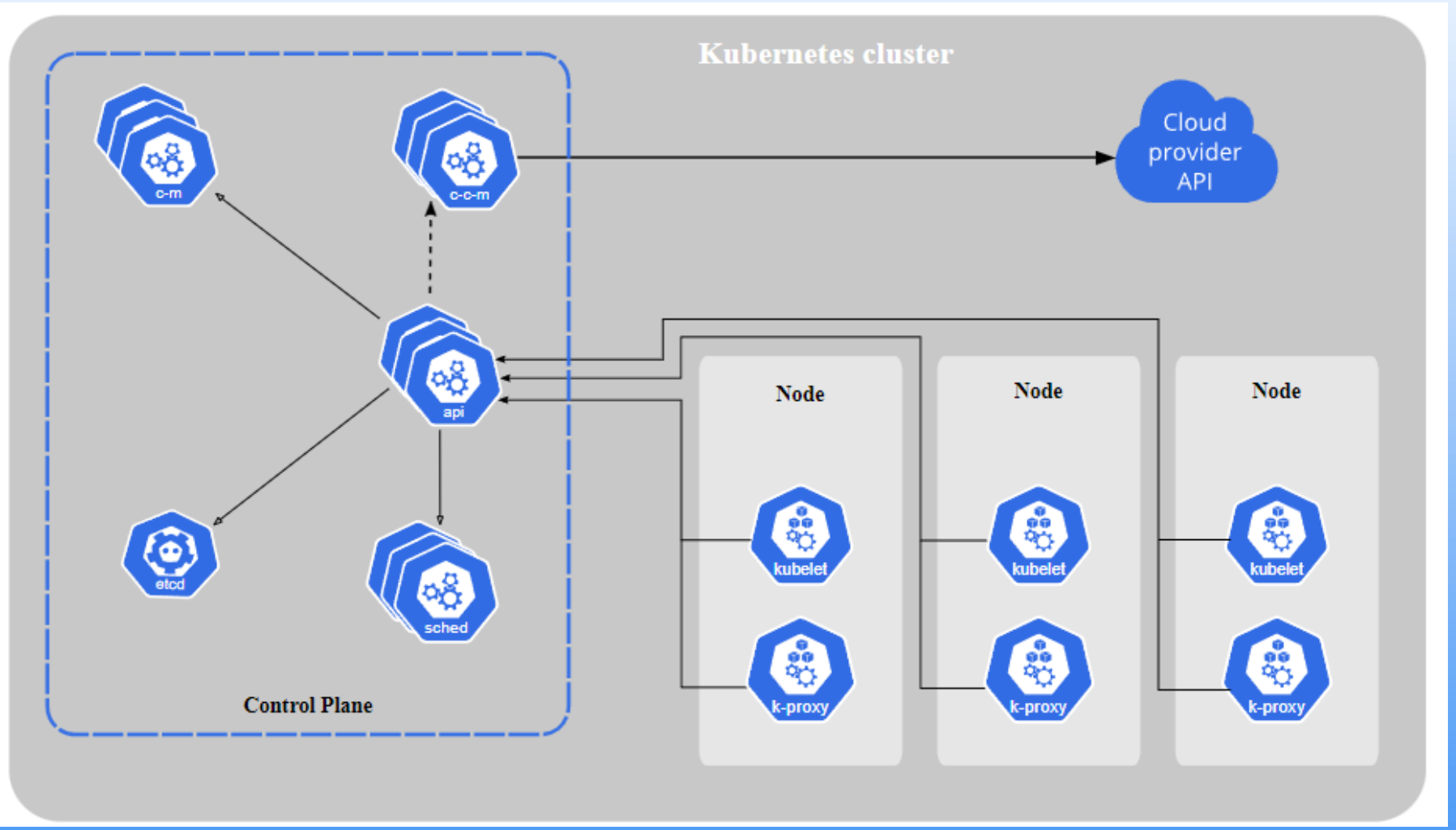
## Kubernetes Components

When you deploy Kubernetes, you get a cluster.

A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node !

The worker node(s) host the Pods that are the components of the application workload.
The control plane manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

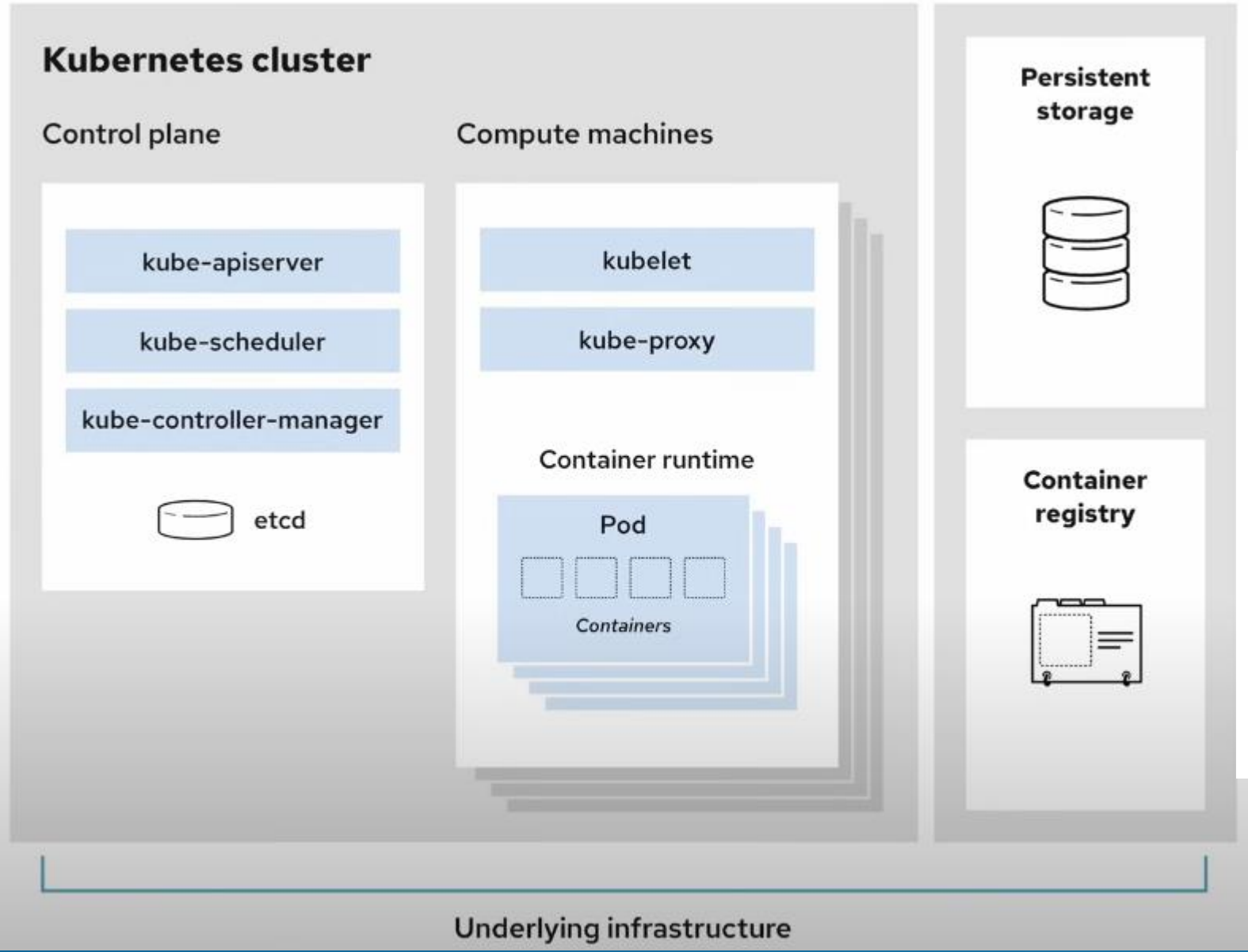
The following outlines the various components you need to have for a complete and working Kubernetes cluster.





**Every cluster needs at least one worker node in order to run Pods !**

*Pods* are the smallest deployable units of computing that you can create and manage in Kubernetes.




Install using native package management

Debian-based distributions | Red Hat-based distributions | SUSE-based distributions

1. Add the Kubernetes `yum` repository. If you want to use Kubernetes version different than v1.33, replace v1.33 with the desired minor version in the command below.

```
# This overwrites any existing configuration in /etc/yum.repos.d/kubernetes.repo
cat <<EOF | sudo tee /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://pkgs.k8s.io/core:/stable:/v1.33/rpm/
enabled=1
gpgcheck=1
gpgkey=https://pkgs.k8s.io/core:/stable:/v1.33/rpm/repodata/repomd.xml.key
EOF
```

**Note:**
To upgrade kubectl to another minor release, you'll need to bump the version in `/etc/yum.repos.d/kubernetes.repo` before running `yum update`. This procedure is described in more detail in Changing The Kubernetes Package Repository.

2. Install kubectl using `yum`:

```
sudo yum install -y kubectl
```

## Control Plane Components

The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new pod when a deployment's replicas field is unsatisfied). Control plane components can be run on any machine in the cluster. However, for simplicity, set up scripts typically start all control plane components on the same machine, and do not run user containers on this machine. See Creating Highly Available clusters with kubeadm for an example control plane setup that runs across multiple machines : stacked control plane nodes *vs* an external etcd cluster.

## kube-apiserver

The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane.

The main implementation of a Kubernetes API server is kube-apiserver. kube-apiserver is designed to scale horizontally—that is, it scales by deploying more instances. You can run several instances of kube-apiserver and balance traffic between those instances.

## etcd

Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

If your Kubernetes cluster uses etcd as its backing store, make sure you have a back up plan for those data. You can find in-depth information about etcd in the official documentation.

**kube-scheduler**

Control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on.

Factors taken into account for scheduling decisions include: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

**kube-controller-manager**

Control plane component that runs controller processes.

Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

Some types of these controllers are:

Node controller: Responsible for noticing and responding when nodes go down.

Job controller: Watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion.

EndpointSlice controller: Populates EndpointSlice objects (to provide a link between Services and Pods).

ServiceAccount controller: Create default ServiceAccounts for new namespaces.

## cloud-controller-manager

A Kubernetes control plane component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with your cluster.

The cloud-controller-manager only runs controllers that are specific to your cloud provider. If you are running Kubernetes on your own premises, or in a learning environment inside your own PC, the cluster does not have a cloud controller manager.

As with the kube-controller-manager, the cloud-controller-manager combines several logically independent control loops into a single binary that you run as a single process. You can scale horizontally (run more than one copy) to improve performance or to help tolerate failures.

The following controllers can have cloud provider dependencies:

- Node controller: For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding
- Route controller: For setting up routes in the underlying cloud infrastructure
- Service controller: For creating, updating and deleting cloud provider load balancers

## Node Components

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

## kubelet

An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod.

The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

## kube-proxy

kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept.

kube-proxy maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.

kube-proxy uses the operating system packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself.

**Container runtime**
The container runtime is the software that is responsible for running containers.
Kubernetes supports container runtimes such as containerd, CRI-O, and any other implementation of the Kubernetes CRI (Container Runtime Interface).

**Addons**
Addons use Kubernetes resources (DaemonSet, Deployment, etc) to implement cluster features.
Because these are providing cluster-level features, namespaced resources for addons belong within the `kube-system` namespace.
Selected addons are described below; for an extended list of available addons, please see Addons.

**DNS**
While the other addons are not strictly required, all Kubernetes clusters should have cluster DNS, as many examples rely on it.
Cluster DNS is a DNS server, in addition to the other DNS server(s) in your environment, which serves DNS records for Kubernetes services.
Containers started by Kubernetes automatically include this DNS server in their DNS searches.

## Web UI (Dashboard)

Dashboard is a general purpose, web-based UI for Kubernetes clusters. It allows users to manage and troubleshoot applications running in the cluster, as well as the cluster itself.

## Container Resource Monitoring

Container Resource Monitoring records generic time-series metrics about containers in a central database, and provides a UI for browsing that data.

## Cluster-level Logging

A cluster-level logging mechanism is responsible for saving container logs to a central log store with search/browsing interface.

# Understanding Kubernetes Objects

This page explains how Kubernetes objects are represented in the Kubernetes API, and how you can express them in `.yaml` format.
*Kubernetes objects* are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of your cluster. Specifically, they can describe:

- What containerized applications are running (and on which nodes)
- The resources available to those applications
- The policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance

A Kubernetes object is a "record of intent"--once you create the object, the Kubernetes system will constantly work to ensure that object exists. By creating an object, you're effectively telling the Kubernetes system what you want your cluster's workload to look like; this is your cluster's *desired state*.

To work with Kubernetes objects - whether to create, modify, or delete them - you'll need to use the Kubernetes API. When you use the `kubectl` command-line interface, for example, the CLI makes the necessary Kubernetes API calls for you. You can also use the Kubernetes API directly in your own programs using one of the Client Libraries.

**Object spec and status**

Almost every Kubernetes object includes two nested object fields that govern the object's configuration: the object *spec* and the object *status*. For objects that have a spec, you have to set this when you create the object, providing a description of the characteristics you want the resource to have: its *desired state*.

The status describes the *current state* of the object, supplied and updated by the Kubernetes system and its components. The Kubernetes control plane continually and actively manages every object's actual state to match the desired state you supplied.

For example: in Kubernetes, a Deployment is an object that can represent an application running on your cluster. When you create the Deployment, you might set the Deployment spec to specify that you want three replicas of the application to be running. The Kubernetes system reads the Deployment spec and starts three instances of your desired application--updating the status to match your spec. If any of those instances should fail (a status change), the Kubernetes system responds to the difference between spec and status by making a correction--in this case, starting a replacement instance.

For more information on the object spec, status, and metadata, see the Kubernetes API Conventions.

# Describing a Kubernetes object

When you create an object in Kubernetes, you must provide the object spec that describes its desired state, as well as some basic information about the object (such as a name). When you use the Kubernetes API to create the object (either directly or via `kubectl`), that API request must include that information as JSON in the request body. **Most often, you provide the information to `kubectl` in a .yaml file.** `kubectl` converts the information to JSON when making the API request.
Here's an example `.yaml` file that shows the required fields and object spec for a Kubernetes Deployment:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

One way to create a Deployment using a `.yaml` file like the one above is to use the `kubectl apply` command in the `kubectl` command-line interface, passing the `.yaml` file as an argument.

Here's an example:

```
kubectl apply -f https://k8s.io/examples/application/deployment.yaml
```

The output is similar to this:

```
deployment.apps/nginx-deployment created
```

**Required fields**
In the `.yaml` file for the Kubernetes object you want to create, you'll need to set values for the following fields:

- **`apiVersion`** - Which version of the Kubernetes API you're using to create this object
- **`kind`** - What kind of object you want to create
- **`metadata`** - Data that helps uniquely identify the object, including a `name` string, `UID`, and optional `namespace`
- **`spec`** - What state you desire for the object

The precise format of the object `spec` is different for every Kubernetes object, and contains nested fields specific to that object. The Kubernetes API Reference can help you find the spec format for all of the objects you can create using Kubernetes.

For example, see the `spec` field for the Pod API reference. For each Pod, the `.spec` field specifies the pod and its desired state (such as the container image name for each container within that pod).

Another example of an object specification is the `spec` field for the StatefulSet API. For StatefulSet, the `.spec` field specifies the StatefulSet and its desired state. Within the `.spec` of a StatefulSet is a template for Pod objects. That template describes Pods that the StatefulSet controller will create in order to satisfy the StatefulSet specification.

Different kinds of object can also have different `.status`; again, the API reference pages detail the structure of that `.status` field, and its content for each different type of object.

# Kubernetes Object Management

The `kubectl` command-line tool supports several different ways to create and manage Kubernetes objects. This document provides an overview of the different approaches. Read the Kubectl book for details of managing objects by Kubectl.

**Management techniques**
**Warning:** A Kubernetes object should be managed using only one technique. Mixing and matching techniques for the same object results in undefined behavior.

| Management technique | Operates on | Recommended environment | Supported writers | Learning curve |
|---|---|---|---|---|
| **Imperative commands** | Live objects | Development projects | 1+ | Lowest |
| **Imperative object configuration** | Individual files | Production projects | 1 | Moderate |
| **Declarative object configuration** | Directories of files | Production projects | 1+ | Highest |

## Imperative commands

When using imperative commands, a user operates directly on live objects in a cluster. The user provides operations to the `kubectl` command as arguments or flags.
This is the recommended way to get started or to run a one-off task in a cluster. Because this technique operates directly on live objects, it provides no history of previous configurations.

*Examples*
Run an instance of the nginx container by creating a Deployment object:

```
kubectl create deployment nginx --image nginx
```

**Trade-offs**
Advantages compared to object configuration:

- Commands are expressed as a single action word.
- Commands require only a single step to make changes to the cluster.

Disadvantages compared to object configuration:

- Commands do not integrate with change review processes.
- Commands do not provide an audit trail associated with changes.
- Commands do not provide a source of records except for what is live.
- Commands do not provide a template for creating new objects.

## Imperative object configuration

In imperative object configuration, the kubectl command specifies the operation (create, replace, etc.), optional flags and at least one file name. The file specified must contain a full definition of the object in YAML or JSON format.

**Warning:** The imperative `replace` command replaces the existing spec with the newly provided one, dropping all changes to the object missing from the configuration file. This approach should not be used with resource types whose specs are updated independently of the configuration file. Services of type `LoadBalancer`, for example, have their `externalIPs` field updated independently from the configuration by the cluster.

*Examples*

Create the objects defined in a configuration file:

```
kubectl create -f nginx.yaml
```

Delete the objects defined in two configuration files:

```
kubectl delete -f nginx.yaml -f redis.yaml
```

Update the objects defined in a configuration file by overwriting the live configuration:

```
kubectl replace -f nginx.yaml
```

**Trade-offs**

Advantages compared to imperative commands:

- Object configuration can be stored in a source control system such as Git.
- Object configuration can integrate with processes such as reviewing changes before push and audit trails.
- Object configuration provides a template for creating new objects.

Disadvantages compared to imperative commands:

- Object configuration requires basic understanding of the object schema.
- Object configuration requires the additional step of writing a YAML file.

Advantages compared to declarative object configuration:

- Imperative object configuration behavior is simpler and easier to understand.
- As of Kubernetes version 1.5, imperative object configuration is more mature.

Disadvantages compared to declarative object configuration:

- Imperative object configuration works best on files, not directories.

Updates to live objects must be reflected in configuration files, or they will be lost during the next replacement.

## Declarative object configuration

When using declarative object configuration, a user operates on object configuration files stored locally, however the user does not define the operations to be taken on the files. Create, update, and delete operations are automatically detected per-object by `kubectl`. This enables working on directories, where different operations might be needed for different objects.

**Note:** Declarative object configuration retains changes made by other writers, even if the changes are not merged back to the object configuration file. This is possible by using the `patch` API operation to write only observed differences, instead of using the `replace` API operation to replace the entire object configuration.

*Examples*

Process all object configuration files in the `configs` directory, and create or patch the live objects. You can first `diff` to see what changes are going to be made, and then apply:

```
kubectl diff -f configs/kubectl apply -f configs/
```

Recursively process directories:

```
kubectl diff -R -f configs/kubectl apply -R -f configs/
```

## Trade-offs

Advantages compared to imperative object configuration:

- Changes made directly to live objects are retained, even if they are not merged back into the configuration files.
- Declarative object configuration has better support for operating on directories and automatically detecting operation types (create, patch, delete) per-object.

Disadvantages compared to imperative object configuration:

- Declarative object configuration is harder to debug and understand results when they are unexpected.
- Partial updates using diffs create complex merge and patch operations.

# Namespaces

In Kubernetes, *namespaces* provide a mechanism for *isolating groups of resources* within a single cluster. Names of resources need to be unique within a namespace, but not across namespaces. Namespace-based scoping is applicable only for namespaced objects *(e.g. Deployments, Services, etc)* and not for cluster-wide objects *(e.g. StorageClass, Nodes, PersistentVolumes, etc)*.

## When to Use Multiple Namespaces

Namespaces are intended for use in environments with many users spread across multiple teams, or projects. For clusters with a few to tens of users, you should not need to create or think about namespaces at all. Start using namespaces when you need the features they provide.
Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces. *Namespaces cannot be nested inside one another and each Kubernetes resource can only be in one namespace*.

Namespaces are a way to divide cluster resources between multiple users (via resource quota).
It is not necessary to use multiple namespaces to separate slightly different resources, such as different versions of the same software: use labels to distinguish resources within the same namespace.

**Note:** For a production cluster, consider *not* using the `default` namespace. Instead, make other namespaces and use those.

**Initial namespaces**
Kubernetes starts with four initial namespaces:

**`default`**
Kubernetes includes this namespace so that you can start using your new cluster without first creating a namespace.

**`kube-node-lease`**
This namespace holds Lease objects associated with each node. Node leases allow the kubelet to send heartbeats so that the control plane can detect node failure.

**`kube-public`**
This namespace is readable by *all* clients (including those not authenticated). This namespace is mostly reserved for cluster usage, in case that some resources should be visible and readable publicly throughout the whole cluster. The public aspect of this namespace is only a convention, not a requirement.

**`kube-system`**
The namespace for objects created by the Kubernetes system.

**Working with Namespaces**
Creation and deletion of namespaces are described in the Admin Guide documentation for namespaces.

**Note:** Avoid creating namespaces with the prefix `kube-`, since it is reserved for Kubernetes system namespaces.

***Viewing namespaces***
You can list the current namespaces in a cluster using:

```
kubectl get namespace
NAME              STATUS   AGE
default           Active   1d
kube-node-lease   Active   1d
kube-public       Active   1d
kube-system       Active   1d
```

***Setting the namespace for a request***
To set the namespace for a current request, use the `--namespace` flag.

For example:
```
kubectl run nginx --image=nginx --namespace=<insert-namespace-name-here>
kubectl get pods --namespace=<insert-namespace-name-here>
```

***Setting the namespace preference***
You can permanently save the namespace for all subsequent kubectl commands in that context.

```
kubectl config set-context --current --namespace=<insert-namespace-name-here>#
Validate it
kubectl config view --minify | grep namespace:
```

***Namespaces and DNS***
When you create a Service, it creates a corresponding DNS entry. This entry is of the form `<service-name>.<namespace-name>.svc.cluster.local`, which means that if a container only uses `<service-name>`, it will resolve to the service which is local to a namespace. This is useful for using the same configuration across multiple namespaces such as Development, Staging and Production. If you want to reach across namespaces, you need to use the fully qualified domain name (FQDN).
As a result, all namespace names must be valid RFC 1123 DNS labels.

Not all objects are in a namespace. Most Kubernetes resources (e.g. pods, services, replication controllers, and others) are in some namespaces. However, namespace resources are not themselves in a namespace. And low-level resources, such as nodes and persistentVolumes, are not in any namespace.

# Field Selectors

*Field selectors* let you select Kubernetes resources based on the value of one or more resource fields. Here are some examples of field selector queries:

- `metadata.name=my-service`
- `metadata.namespace!=default`
- `status.phase=Pending`

This `kubectl` command selects all Pods for which the value of the `status.phase` field is `Running`:

```
kubectl get pods --field-selector status.phase=Running
```

**Note:** Field selectors are essentially resource *filters*. By default, no selectors/filters are applied, meaning that all resources of the specified type are selected. This makes the `kubectl` queries `kubectl get pods` and `kubectl get pods --field-selector ""` equivalent.

**Supported fields**

Supported field selectors vary by Kubernetes resource type. All resource types support the `metadata.name` and `metadata.namespace` fields. Using unsupported field selectors produces an error. For example:

```
kubectl get ingress --field-selector foo.bar=bazError from server (BadRequest): Unable to find "ingresses" that match label selector "", field selector "foo.bar=baz": "foo.bar" is not a known field selector: only "metadata.name", "metadata.namespace"
```

**Supported operators**

You can use the `=`, `==`, and `!=` operators with field selectors (`=` and `==` mean the same thing).

This `kubectl` command, for example, selects all Kubernetes Services that aren't in the `default` namespace:

```
kubectl get services  --all-namespaces --field-selector metadata.namespace!=default
```

To illustrate these labels in action, consider the following StatefulSet object:

```yaml
# This is an excerpt
apiVersion: apps/v1
kind: StatefulSet
metadata:
       labels:
          app.kubernetes.io/name: mysql
          app.kubernetes.io/instance: mysql-abcxzy
          app.kubernetes.io/version: "5.7.21"
          app.kubernetes.io/component: database
          app.kubernetes.io/part-of: wordpress
          app.kubernetes.io/managed-by: helm
```

# Applications And Instances Of Applications

An application can be installed one or more times into a Kubernetes cluster and, in some cases, the same namespace.
For example, WordPress can be installed more than once where different websites are different installations of WordPress.

The name of an application and the instance name are recorded separately.

For example, WordPress has a `app.kubernetes.io/name` of `wordpress` while it has an instance name, represented as `app.kubernetes.io/instance` with a value of `wordpress-abcxzy`.

This enables the application and instance of the application to be identifiable.

Every instance of an application must have a unique name.

# Examples

To illustrate different ways to use these labels the following examples have varying complexity.

### *A Simple Stateless Service*

Consider the case for a simple stateless service deployed using `Deployment` and `Service` objects. The following two snippets represent how the labels could be used in their simplest form.
The `Deployment` is used to oversee the pods running the application itself.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/name: myservice
    app.kubernetes.io/instance: myservice-abcxzy...
```

The `Service` is used to expose the application.

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/name: myservice
    app.kubernetes.io/instance: myservice-abcxzy
...
```

***Web Application With A Database***

Consider a slightly more complicated application: a web application (WordPress) using a database (MySQL), installed using Helm.
The following snippets illustrate the start of objects used to deploy this application.

The start to the following `Deployment` is used for WordPress:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/name: wordpress
    app.kubernetes.io/instance: wordpress-abcxzy
    app.kubernetes.io/version: "4.9.4"
    app.kubernetes.io/managed-by: helm
    app.kubernetes.io/component: server
    app.kubernetes.io/part-of: wordpress
...
```

The `Service` is used to expose WordPress:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/name: wordpress
    app.kubernetes.io/instance: wordpress-abcxzy
    app.kubernetes.io/version: "4.9.4“
    app.kubernetes.io/managed-by: helm
    app.kubernetes.io/component: server
    app.kubernetes.io/part-of: wordpress
...
```

MySQL is exposed as a `StatefulSet` with metadata for both it and the larger application (wordpress) it belongs to:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  labels:
    app.kubernetes.io/name: mysql
    app.kubernetes.io/instance: mysql-abcxzy
    app.kubernetes.io/version: "5.7.21“
    app.kubernetes.io/managed-by: helm
    app.kubernetes.io/component: database
    app.kubernetes.io/part-of: wordpress
...
```

The `Service` is used to expose MySQL as part of WordPress:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/name: mysql
    app.kubernetes.io/instance: mysql-abcxzy
    app.kubernetes.io/version: "5.7.21“
    app.kubernetes.io/managed-by: helm
    app.kubernetes.io/component: database
    app.kubernetes.io/part-of: wordpress
...
```

With the MySQL `StatefulSet` and `Service` you'll notice information about both MySQL and WordPress, the broader application, are included.

# Workloads

**A workload is an application running on Kubernetes**. Whether your workload is a single component or several that work together, on Kubernetes you run it inside a set of *pods*. In Kubernetes, a **Pod** represents a set of running containers on your cluster.

Kubernetes pods have a defined lifecycle. For example, once a pod is running in your cluster then a critical fault on the node where that pod is running means that all the pods on that node fail. Kubernetes treats that level of failure as final: you would need to create a new Pod to recover, even if the node later becomes healthy.

However, to make life considerably easier, you don't need to manage each Pod directly. Instead, you can use *workload resources* that manage a set of pods on your behalf. These resources configure controllers that make sure the right number of the right kind of pod are running, to match the state you specified.

Kubernetes provides several built-in workload resources:

- `Deployment` and `ReplicaSet` (replacing the legacy resource ReplicationController). `Deployment` is a good fit for managing a stateless application workload on your cluster, where any `Pod` in the `Deployment` is interchangeable and can be replaced if needed.

- `StatefulSet` lets you run one or more related Pods that do track state somehow. For example, if your workload records data persistently, you can run a `StatefulSet` that matches each `Pod` with a `PersistentVolume`. Your code, running in the `Pods` for that `StatefulSet`, can replicate data to other `Pods` in the same `StatefulSet` to improve overall resilience.

- `DaemonSet` defines `Pods` that provide node-local facilities. These might be fundamental to the operation of your cluster, such as a networking helper tool, or be part of an add-on. Every time you add a node to your cluster that matches the specification in a `DaemonSet`, the control plane schedules a `Pod` for that `DaemonSet` onto the new node.

- `Job` and `CronJob` define tasks that run to completion and then stop. Jobs represent one-off tasks, whereas `CronJobs` recur according to a schedule.

## Run a Stateless Application Using a Deployment

This page shows how to run an application using a Kubernetes Deployment object.

## Objectives

- Create an nginx deployment.
- Use kubectl to list information about the deployment.
- Update the deployment.

## Creating and exploring a nginx deployment

You can run an application by creating a Kubernetes Deployment object, and you can describe a Deployment in a YAML file. For example, this YAML file describes a Deployment that runs the nginx:1.14.2 Docker image.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Create a Deployment based on the YAML file:

1. Get the .yaml configuration file of your deployment and run it:
   ```
   kubectl apply -f https://k8s.io/examples/application/deployment.yaml
   ```

2. Display information about the Deployment:
   ```
   kubectl describe deployment nginx-deployment
   ```

   The output is similar to this:

```
Name:     nginx-deployment
Namespace:    default
CreationTimestamp:  Tue, 30 Aug 2016 18:11:37 -0700
Labels:     app=nginx
Annotations:    deployment.kubernetes.io/revision=1
Selector:   app=nginx
Replicas:   2 desired | 2 updated | 2 total | 2 available | 0 unavailable
StrategyType:   RollingUpdate
MinReadySeconds:  0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
Pod Template:
  Labels:       app=nginx
  Containers:
    nginx:
    Image:              nginx:1.14.2
    Port:               80/TCP
    Environment:        <none>
    Mounts:             <none>
  Volumes:              <none>
Conditions:
  Type          Status  Reason
  ----          ------  ------
  Available     True    MinimumReplicasAvailable
  Progressing   True    NewReplicaSetAvailable
OldReplicaSets:   <none>
NewReplicaSet:    nginx-deployment-1771418926 (2/2 replicas created)
No events.
```

3. List the Pods created by the deployment:
   ```
   kubectl get pods -l app=nginx
   ```

   The output is similar to this:

```
NAME                                READY     STATUS    RESTARTS   AGE
nginx-deployment-1771418926-7o5ns   1/1       Running   0          16h
nginx-deployment-1771418926-r18az   1/1       Running   0          16h
```

4. Display information about a Pod:
   ```
   kubectl describe pod <pod-name>
   ```

   where `<pod-name>` is the name of one of your Pod(s).

## Updating the deployment

You can update the deployment by applying a new YAML file.
This YAML file specifies that the deployment should be updated to use nginx 1.16.1:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.16.1 # Update the version of nginx from 1.14.2 to 1.16.1
      ports:
        - containerPort: 80
```

1. Apply the new YAML file:

```
kubectl apply -f https://k8s.io/examples/application/deployment-update.yaml
```

2. Watch the deployment create pods with new names and delete the old pods:

```
kubectl get pods -l app=nginx
```

## Scaling the application by increasing the replica count

You can increase the number of Pods in your Deployment by applying a new YAML file.

This YAML file sets `replicas` to 4, which specifies that the Deployment should have four Pods:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 4 # Update the replicas from 2 to 4
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.16.1
        ports:
        - containerPort: 80
```

1. Apply the new YAML file:

```
kubectl apply -f https://k8s.io/examples/application/deployment-scale.yaml
```

2. Verify that the Deployment has four Pods:

```
kubectl get pods -l app=nginx
```

The output is similar to this:

```
NAME                                READY     STATUS    RESTARTS   AGE
nginx-deployment-148880595-4zdqq    1/1       Running   0          25s
nginx-deployment-148880595-6zgi1    1/1       Running   0          25s
nginx-deployment-148880595-fxcez    1/1       Running   0          2m
nginx-deployment-148880595-rwovn    1/1       Running   0          2m
```

## Deleting a deployment

Delete the deployment by name:

```
kubectl delete deployment nginx-deployment
```

# Run a Single-Instance Stateful Application

This page shows you how to run a single-instance stateful application in Kubernetes using a PersistentVolume and a Deployment. The application is MySQL.

Objectives:

- Create a PersistentVolume referencing a disk in your environment.
- Create a MySQL Deployment.
- Expose MySQL to other pods in the cluster at a known DNS name.

## Deploy MySQL

You can run a stateful application by creating a Kubernetes Deployment and connecting it to an existing *PersistentVolume* using a *PersistentVolumeClaim (PVC)*. For example, this YAML file describes a Deployment that runs MySQL and references the *PersistentVolumeClaim*.
The file defines a volume mount for */var/lib/mysql*, and then creates a *PersistentVolumeClaim* that looks for a 20G volume. This claim is satisfied by any existing volume that meets the requirements, or by a dynamic provisioner (i.e., longhorn)

Note: The password is defined in the config yaml, and this is insecure. See Kubernetes Secrets for a secure solution.

## MySQL service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mysql
spec:
  ports:
  - port: 3306
  selector:
    app: mysql
  clusterIP: None

...
```

## MySQL deployment

```yaml
...
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - image: mysql:5.6
        name: mysql
        env:          # Use secret in real usage
        - name: MYSQL_ROOT_PASSWORD
          value: password
        ports:
        - containerPort: 3306
          name: mysql
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-persistent-storage
        persistentVolumeClaim:
          claimName: mysql-pv-claim
```

## Creating PV

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 20Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"

...
```

## Creating a PVC

```yaml
...

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
```

1. Deploy the PV and PVC of the YAML file:

```
kubectl apply -f https://k8s.io/examples/application/mysql/mysql-pv.yaml
```

2. Deploy the contents of the YAML file:

```
kubectl apply -f https://k8s.io/examples/application/mysql/mysql-deployment.yaml
```

3. Display information about the Deployment:

```
kubectl describe deployment mysql
```

The output is similar to **[A]**:

4. List the pods created by the Deployment:

```
kubectl get pods -l app=mysql
```

The output is similar to this:

```
NAME                    READY     STATUS    RESTARTS   AGE
mysql-63082529-2z3ki    1/1       Running   0          3m
```

5. Inspect the PersistentVolumeClaim:

```
kubectl describe pvc mysql-pv-claim
```

The output is similar to this:

```
Name:          mysql-pv-claim
Namespace:     default
StorageClass:
Status:        Bound
Volume:        mysql-pv-volume
Labels:        <none>
Annotations:   pv.kubernetes.io/bind-completed=yes
               pv.kubernetes.io/bound-by-controller=yes
Capacity:      20Gi
```

# [A]

```
Name:                   mysql
Namespace:              default
CreationTimestamp:      Tue, 01 Nov 2016 11:18:45 -0700
Labels:                 app=mysql
Annotations:            deployment.kubernetes.io/revision=1
Selector:               app=mysql
Replicas:               1 desired | 1 updated | 1 total | 0 available | 1 unavailable
StrategyType:           Recreate
MinReadySeconds:        0
Pod Template:
  Labels:       app=mysql
  Containers:
    mysql:
    Image:      mysql:5.6
    Port:       3306/TCP
    Environment:        MYSQL_ROOT_PASSWORD:      password
    Mounts:      /var/lib/mysql from mysql-persistent-storage (rw)
  Volumes:    mysql-persistent-storage:
    Type:       PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the same namespace)
    ClaimName:  mysql-pv-claim
    ReadOnly:   false
Conditions:
  Type          Status  Reason
  ----          ------  ------
  Available     False   MinimumReplicasUnavailable
  Progressing   True    ReplicaSetUpdated
OldReplicaSets:         <none>
NewReplicaSet:          mysql-63082529 (1/1 replicas created)
Events:
  FirstSeen    LastSeen    Count    From                SubobjectPath    Type        Reason            Message
  ---------    --------    -----    ----                -------------    --------    ------            -------
  33s          33s         1        {deployment-controller }             Normal      ScalingReplicaSet Scaled up replica set mysql-63082529 to 1
```

**Accessing the MySQL instance**

The preceding YAML file creates a service that allows other Pods in the cluster to access the database. The Service option `clusterIP: None` lets the Service DNS name resolve directly to the Pod's IP address. This is optimal when you have only one Pod behind a Service and you don't intend to increase the number of Pods.

**Run a MySQL client to connect to the server:**

```
kubectl run -it --rm --image=mysql:5.6 --restart=Never mysql-client -- mysql -h mysql -ppassword
```

This command creates a new Pod in the cluster running a MySQL client and connects it to the server through the Service. If it connects, you know your stateful MySQL database is up and running.

```
Waiting for pod default/mysql-client-274442439-zyp6i to be running, status is Pending,
pod ready: false
If you don't see a command prompt, try pressing enter.

mysql>
```

# Updating

The image or any other part of the Deployment can be updated as usual with the `kubectl apply` command. Here are some precautions that are specific to stateful apps:

- Don't scale the app. This setup is for single-instance apps only. The underlying PersistentVolume can only be mounted to one Pod. For clustered stateful apps, see the StatefulSet documentation.
- Use `strategy: type: Recreate` in the Deployment configuration YAML file. This instructs Kubernetes to *not* use rolling updates. Rolling updates will not work, as you cannot have more than one Pod running at a time. The `Recreate` strategy will stop the first pod before creating a new one with the updated configuration.

# Deleting a deployment

Delete the deployed objects by name:

```
kubectl delete deployment,svc mysql
kubectl delete pvc mysql-pv-claim
kubectl delete pv mysql-pv-volume
```

If you manually provisioned a *PersistentVolume*, you also need to manually delete it, as well as release the underlying resource. If you used a dynamic provisioner, it automatically deletes the *PersistentVolume* when it sees that you deleted the *PersistentVolumeClaim*. Some dynamic provisioners (such as Longhorn we have in Production cluster) also release the underlying resource upon deleting the *PersistentVolume*.

# Creating a CronJob

Cron jobs require a config file. Here is a manifest for a CronJob that runs a simple demonstration task every minute:

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "* * * * *“
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox:1.28
            imagePullPolicy: IfNotPresent
            command:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

Run the example CronJob by using this command:

```
kubectl create -f https://k8s.io/examples/application/job/cronjob.yaml
```

The output is similar to this:

```
cronjob.batch/hello created
```

After creating the cron job, get its status using this command:

```
kubectl get cronjob hello
```

The output is similar to this:

```
NAME    SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
hello   */1 * * * *   False     0        <none>          10s
```

As you can see from the results of the command, the cron job has not scheduled or run any jobs yet.

Watch for the job to be created in around one minute:

```
kubectl get jobs --watch
```

The output is similar to this:

```
NAME                COMPLETIONS   DURATION   AGE
hello-4111706356    0/1                      0s
hello-4111706356    0/1           0s         0s
hello-4111706356    1/1           5s         5s
```

Now you've seen one running job scheduled by the "hello" cron job.
You can stop watching the job and view the cron job again to see that it scheduled the job:
`kubectl get cronjob hello`

The output is similar to this:
```
NAME    SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
hello   */1 * * * *   False     0        50s             75s
```

You should see that the cron job `hello` successfully scheduled a job at the time specified in `LAST SCHEDULE`.
There are currently 0 active jobs, meaning that the job has completed or failed.
Now, find the pods that the last scheduled job created and view the standard output of one of the pods.
**Note:** The job name is different from the pod name.
```
# Replace "hello-4111706356" with the job name in your system
pods=$(kubectl get pods --selector=job-name=hello-4111706356 --output=jsonpath={.items[*].metadata.name})
```

Show the pod log:
```
kubectl logs $pods
```

The output is similar to this:
```
Fri Feb 22 11:02:09 UTC 2019
Hello from the Kubernetes cluster
```

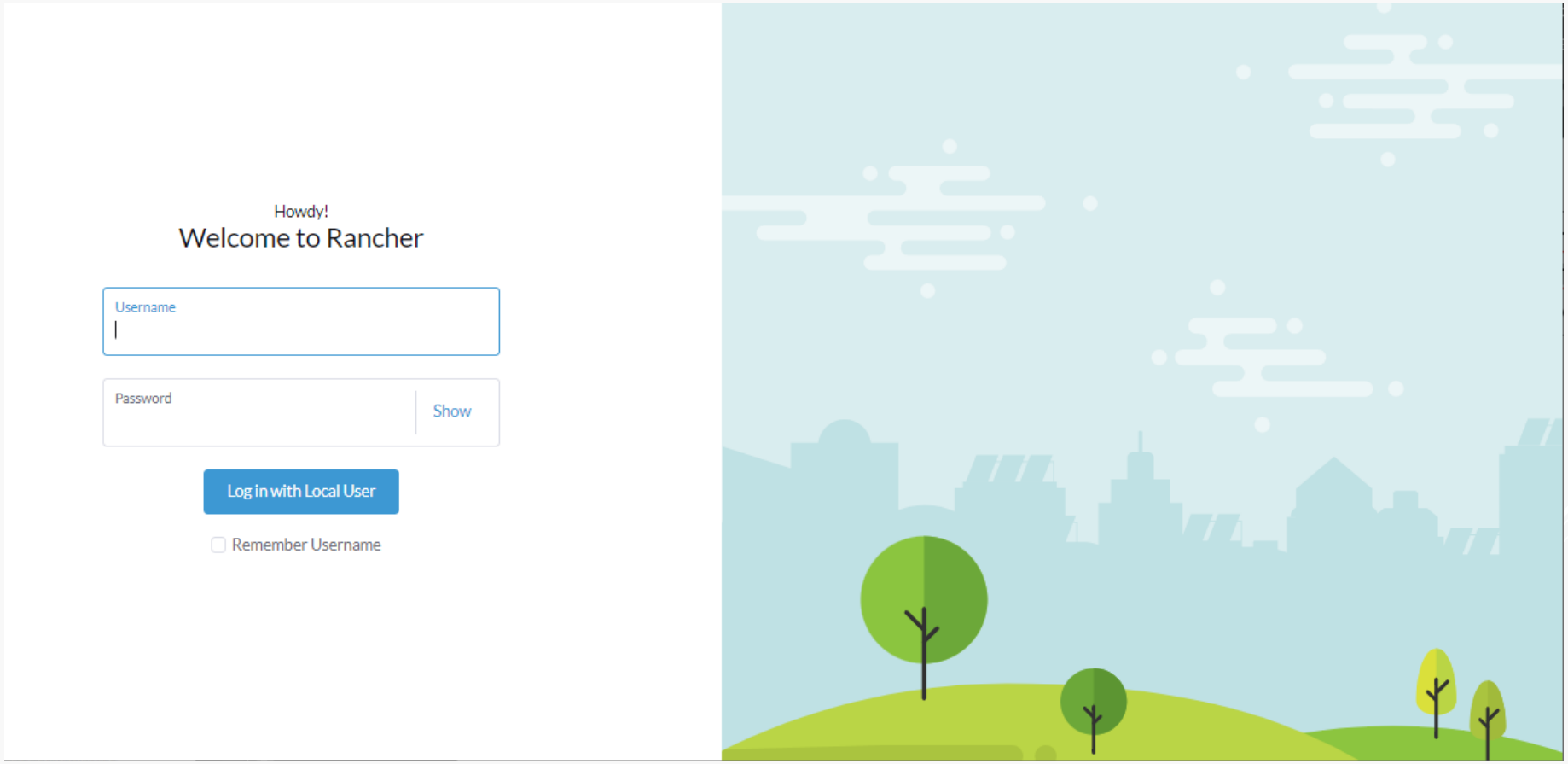
# Deleting a CronJob

When you don't need a cron job anymore, delete it with `kubectl delete cronjob <cronjob name>`:
```
kubectl delete cronjob hello
```

Deleting the cron job removes all the jobs and pods it created and stops it from creating additional jobs.
You can read more about removing jobs in garbage collection.

Lets stop here our introductory mini-course on kubernetes and come on into the hands-on by practicing with kubectl and simplify the interaction with the kube cluster thanks to Rancher!!!

https://narten.unitus.it:5019




NOTE: For those wanting more on kubernetes, please have a look to file *Kubernets_AdvancedTopics.pdf* in our gdrive ☺