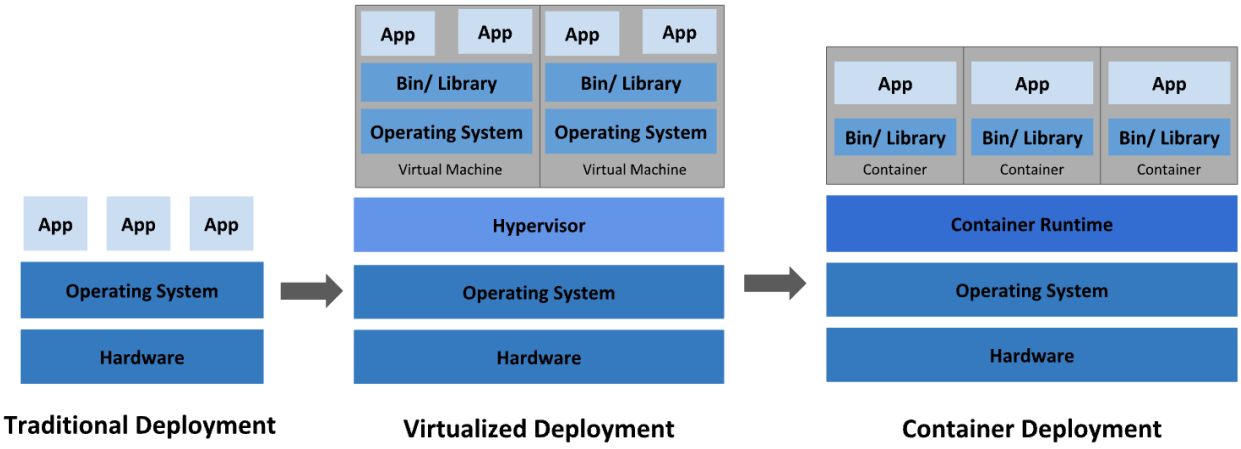
Containers

Virtual Machine VS Containers



Each virtual machine contains a guest OS, a virtual copy of the hardware that the OS requires to run and an application and its associated libraries and dependencies.

Instead of virtualizing the underlying hardware, containers virtualize the operating system (typically Linux or Windows) so each individual container contains only the application and its libraries and dependencies. Containers are small, fast, and portable because, unlike a virtual machine, containers do not need to include a guest OS in every instance and can, instead, simply leverage the features and resources of the host OS.

Just like virtual machines, containers allow developers to improve CPU and memory utilization of physical machines. Containers go even further, however, because they also enable microservice architectures, where application components can be deployed and scaled more granularly. This is an attractive alternative to having to scale up an entire monolithic application because a single component is struggling with load.

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 benefits:

- 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.

Container tools

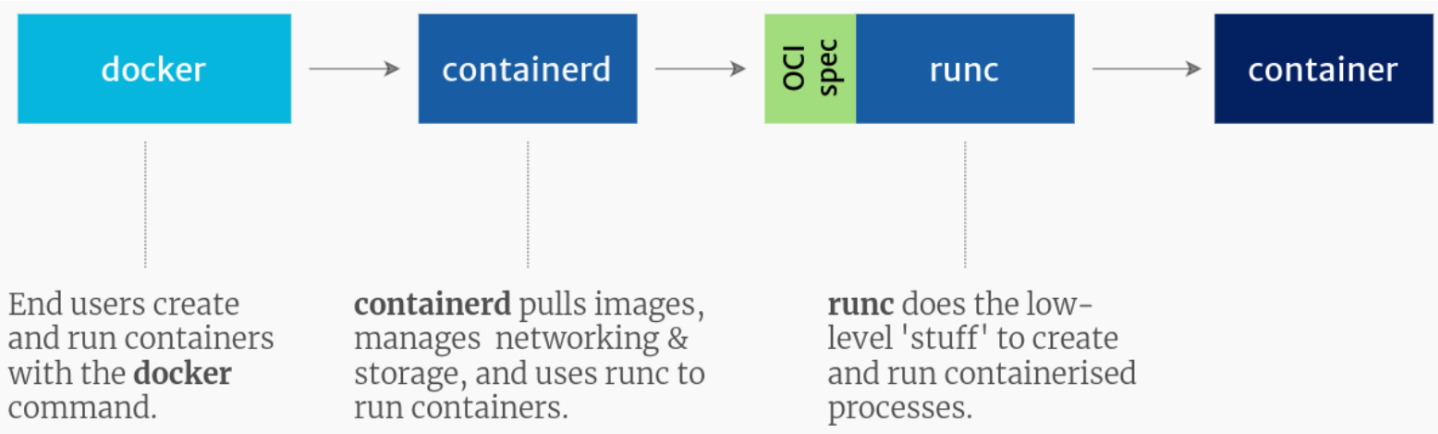
https://www.tutorialworks.com/difference-docker-containerd-runc-crio-oci/

Containers are no longer tightly coupled with the name Docker. There is a whole set of container tools out there, docker being one of them, and Docker (the company) backing some of them, but not all.

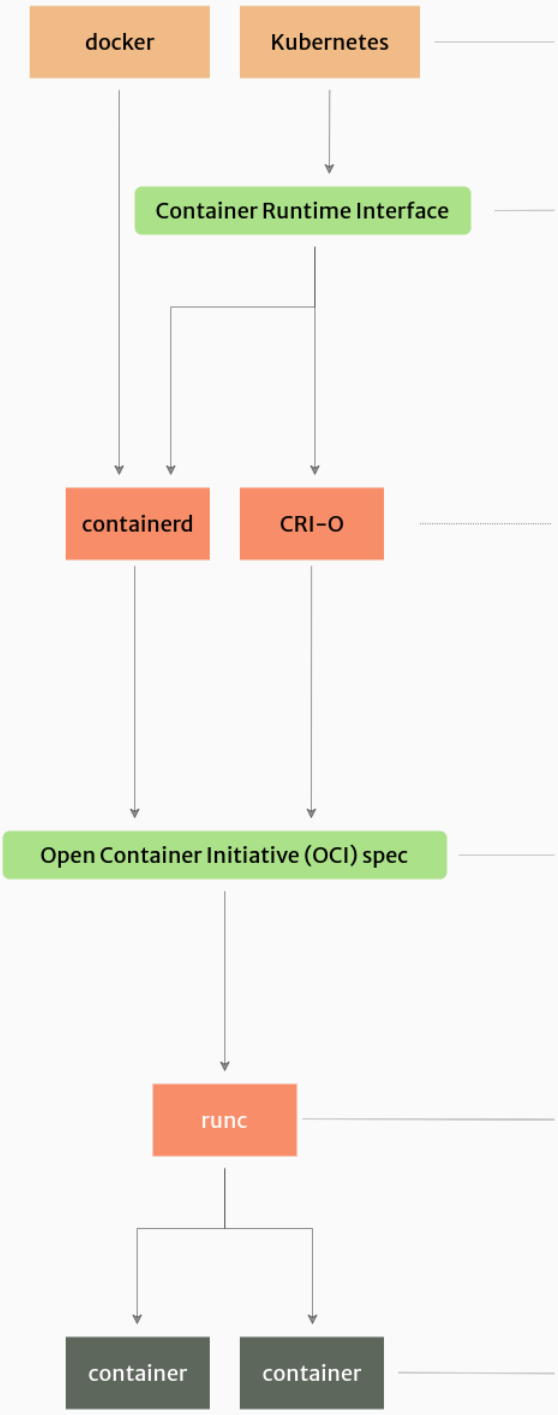
The container ecosystem is made up of lots of exciting tech, plenty of jargon, and big companies fighting each other. Fortunately, these companies occasionally come together in a fragile truce to agree to some standards. These standards help to make the ecosystem more interoperable, across different platforms and operating systems, and less reliant on one single company or project.

The main standards to be aware of are:

- the **Kubernetes Container Runtime Interface** (CRI), which defines an API between Kubernetes and the container runtime
- the **Open Container Initiative** (OCI) which publishes specifications for images and containers.



Docker, Kubernetes, OCI, CRI-O, containerd & runc: **How do they work together?**



These are the tools that you use to run containers, as a developer, or in production.

CRI is a Kubernetes API.

It defines the way that Kubernetes interacts with different container runtimes.

Because it's standardised in a spec, so you can choose which CRI implementation you want to use. Or write your own.

You can choose from a couple of runtimes that implement the CRI spec.

containerd came from Docker. It's made CRI-compliant through its *cri* plugin.

CRI-O was developed as a CRI implementation, from Red Hat/IBM/etc.

OCI provides specifications for container images and running containers.

It also provides an implementation of the spec, *runc*. It is a tool for creating and running container processes.

runc is an OCI-compliant tool for spawning and running containers.

It implements the OCI spec, and is responsible for creating and running the container processes.

Finally, your containerised process is running!

Docker

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow you to run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you do not need to rely on what is currently installed on the host. You can easily share containers while you work, and be sure that everyone you share with gets the same container that works in the same way.

Docker provides tooling and a platform to manage the lifecycle of your containers:

- **Develop** your application and its supporting components using containers.
- The container becomes the **unit for distributing and testing** your application.
- When you're ready, **deploy** your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

docker is designed to be installed on a workstation or server and comes with a bunch of tools to make it easy to build and run containers as a developer, or DevOps person.

The docker command line tool can build container images, pull them from registries, create, start and manage containers.

To make all of this happen, the experience you know as docker is now comprised of these projects (there are others, but these are the main ones):

- **docker-cli**: This is the command-line utility that you interact with using **docker** commands. It is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon
- **dockerd**: The docker high-level daemon listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.
- **containerd**: This is a daemon process listening on a Unix socket, exposes gRPC endpoints. It manages and runs containers. Handles low-level container management tasks - pushes and pulls images, manages storage and networking, and supervises the running of containers.
- **runc**: This is the low-level container runtime (the thing that actually creates and runs containers). It includes libcontainer, a native Go-based implementation for creating containers. It deals with the low-level interfacing with Linux capabilities like cgroups, namespaces, etc.

In reality, when you run a container with docker, you're actually running it through the Docker daemon, containerd, and then runc.

Other docker components that are worth mentioning:

- (docker-)**containerd-ctr** - A lightweight CLI to directly communicate with containerd. Think of it as how 'docker' is to 'dockerd'.
- (docker-)**containerd-shim** - After runC actually runs the container, it exits (allowing us to not have any long-running processes responsible for our containers). The shim is the component which sits between containerd and runc to facilitate this.

The underlying technology

Docker is written in the **Go programming language** and takes advantage of several features of the Linux kernel to deliver its functionality. Docker uses a technology called **namespaces** to provide the isolated workspace called the container. When you run a container, Docker creates a set of namespaces for that container.

These namespaces provide a layer of isolation. Each aspect of a container runs in a separate namespace and its access is limited to that namespace.

Docker Image

https://www.scalyr.com/blog/create-docker-image/

A Docker image is a file used to execute code in a Docker container. Docker images act as a **set of instructions** to build a Docker container, like a template. Docker images also act as the starting point when using Docker. An **image is comparable to a snapshot in virtual machine (VM)** environments.

Docker is used to create, run and deploy applications in containers. A Docker image contains **application code, libraries, tools, dependencies and other files needed to make an application run**. When a user runs an image, it can become one or many instances of a container.

A Docker image is like a snapshot in other types of VM environments. It is a **record of a Docker container** at a specific point in time.

Docker images are also immutable. While they can't be changed, they can be duplicated, shared or deleted. The feature is useful for testing new software or configurations because whatever happens, the image remains unchanged.

A Docker image has many layers, and each image includes everything needed to configure a container environment - system libraries, tools, dependencies and other files. Some of the parts of an image include:

- **Base image**. The user can build this first layer entirely from scratch with the build command.
- **Parent image**. As an alternative to a base image, a parent image can be the first layer in a Docker image. It is a reused image that serves as a foundation for all other layers.
- **Layers**. Layers are added to the base image, using code that will enable it to run in a container. Each layer of a Docker image is viewable under /var/lib/docker/aufs/diff, or via the Docker history command in the command-line interface (CLI). Docker's default status is to show all top-layer images, including repository, tags and file sizes. Intermediate layers are cached, making top layers easier to view. Docker has storage drives that handle the management of image layer contents.
- **Container layer**. A Docker image not only creates a new container, but also a writable or container layer. This layer hosts changes made to the running container and stores newly written and deleted files, as well as changes to existing files. This layer is also used to customize containers.
- **Docker manifest**. This part of the Docker image is an additional file. It uses JSON format to describe the image, using information such as image tags and digital signature.

There are two ways to create a Docker Image:

- **Interactive method** - users run a container from an existing Docker image and manually make any needed changes to the environment before saving the image with **docker commit**.
- **Dockerfile method** - this approach requires making a plain text Dockerfile. The Dockerfile makes the **specifications** for creating an image. This process is more difficult and time-consuming, but it does well in continuous delivery environments. The method includes creating the Dockerfile and adding the

commands needed for the image. Once the Dockerfile is started, the user sets up a .dockerignore file to exclude any files not needed for the final build. The .dockerignore file is in the root directory. Next, the **docker build** command is used to create a Docker image and an image name and tag are set.

A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry. When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

The concept of a **latest** image may cause confusion. Docker images tagged with ":latest" are not necessarily the latest in an ordinary sense. The latest tag does not refer to the most recently pushed version of an image; it is simply a default tag.

Docker Container

A container is a **runnable instance of an image**. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.

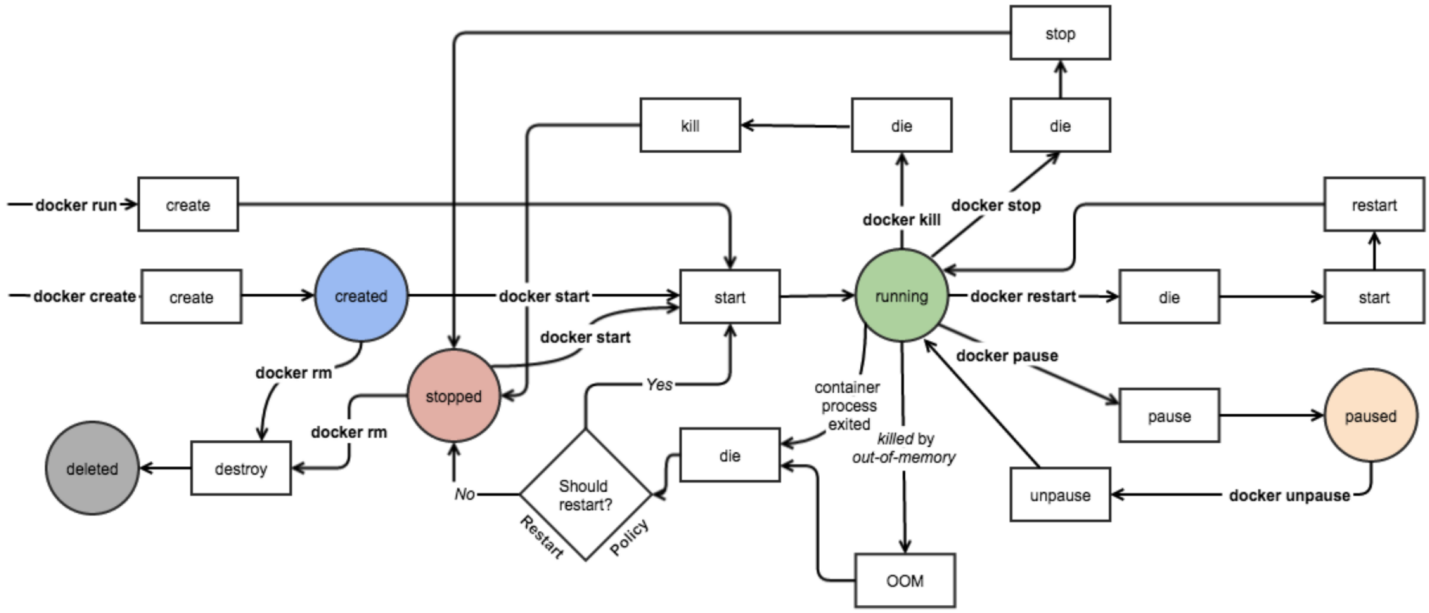
A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

The following command runs an ubuntu container, attaches interactively to your local command-line session, and runs /bin/bash:

```
docker run -i -t ubuntu /bin/bash
```

When you run this command, the following happens (assuming you are using the default registry configuration):

- If you do not have the ubuntu image locally, Docker **pulls** it from your configured registry, as though you had run docker pull ubuntu manually.
- Docker **creates a new container**, as though you had run a docker container create command manually.
- Docker **allocates a read-write filesystem** to the container, as its final layer. This allows a running container to create or modify files and directories in its local filesystem.
- Docker **creates a network interface** to connect the container to the default network, since you did not specify any networking options. This includes assigning an IP address to the container. By default, containers can connect to external networks using the host machine's network connection.
- Docker **starts the container and executes /bin/bash**. Because the container is running interactively and attached to your terminal (due to the -i and -t flags), you can provide input using your keyboard while the output is logged to your terminal.
- When you type exit to terminate the /bin/bash command, the container **stops but is not removed**. You can start it again or remove it.



Runtime options

By default, a **container has no resource constraints** and can use as much of a given resource as the host's kernel scheduler allows. Docker provides ways to control how much memory, or CPU a container can use, setting runtime configuration flags of the docker **run** command, some of them are below:

-m or --memory	The maximum amount of memory the container can use. If you set this option, the minimum allowed value is 6m (6 megabyte)
--memory-reservation	Allows you to specify a **soft limit** smaller than --memory which is activated when Docker detects contention or low memory on the host machine. If you use --memory-reservation, it must be set lower than --memory for it to take precedence. Because it is a soft limit, it does not guarantee that the container doesn't exceed the limit.
--cpus	Specify how much of the available CPU resources a container can use. For instance, if the host machine has two CPUs and you set --cpus="1.5", the container is guaranteed at most one and a half of the CPUs. This is the equivalent of setting --cpu-period="100000" and --cpu-quota="150000"
--cpu-shares	Set this flag to a value greater or less than the default of 1024 to increase or reduce the **container's weight**, and give it access to a greater or lesser proportion of the host machine's CPU cycles. This is only enforced when CPU cycles are constrained. When plenty of CPU cycles are available, all containers use as much CPU as they need. In that way, this is a **soft limit**

Docker-compose

When using Docker extensively, the management of several different containers quickly becomes cumbersome.

Docker Compose is a tool for defining and running **multi-container Docker applications**. With Compose, you use a **YAML** file to configure your application's services. Then, with a single command, you create and start all the services from your configuration. To learn more about all the features of Compose, see the list of features.

Using Compose is basically a three-step process:

- Define your app's environment with a **Dockerfile** so it can be reproduced anywhere.
- Define the services that make up your app in **docker-compose.yml** so they can be run together in an isolated environment.
- Run **docker compose up** and the Docker compose command starts and runs your entire app. You can alternatively run **docker-compose up** using the docker-compose binary.

A docker-compose.yml looks like this:

```
version: "3.9"  # optional since v1.27.0
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
      - logvolume01:/var/log
    links:
      - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

Compose has commands for managing the whole lifecycle of your application:

- Start, stop, and rebuild services
- View the status of running services
- Stream the log output of running services
- Run a one-off command on a service

Compose preserves all volumes used by your services. When **docker-compose up** runs, if it finds any containers from previous runs, it **copies the volumes from the old container to the new container**. This process ensures that any data you've created in volumes isn't lost.

Only recreate containers that have changed: compose caches the configuration used to create a container. When you restart a service that has not changed, Compose **re-uses the existing containers**. Re-using containers means that you can make changes to your environment very quickly

Common use cases

- **Development environments**: When you're developing software, the ability to run an application in an isolated environment and interact with it is crucial. The Compose command line tool can be used to create the environment and interact with it.

  The Compose file provides a way to document and configure all of the application's service dependencies (databases, queues, caches, web service APIs, etc). Using the Compose command line tool you can create and start one or more containers for each dependency with a single command (docker-compose up).

  Together, these features provide a convenient way for developers to get started on a project. Compose can reduce a multi-page "developer getting started guide" to a single machine readable Compose file and a few commands.

- **Automated testing environments**: An important part of any Continuous Deployment or Continuous Integration process is the automated test suite. Automated end-to-end testing requires an environment in which to run tests. Compose provides a convenient way to create and destroy isolated testing environments for your test suite. By defining the full environment in a Compose file, you can create and destroy these environments in just a few commands:

  ```
  docker-compose up -d
  ./run_tests
  docker-compose down
  ```

Service

Sometimes, the image we need for our service has already been published (by us or by others) in Docker Hub, or another Docker Registry. If that's the case, then we refer to it with the image attribute, by specifying the image name and tag:

```
services:
  my-service:
    image: ubuntu:latest
    ...
```

Instead, we might need to build an image from the source code by reading its Dockerfile. This time, we'll use the build keyword, passing the path to the Dockerfile as the value:

```
services:
  my-custom-app:
    build: /path/to/dockerfile/
    ...
```

We can also use a URL instead of a path:

```
services:
  my-custom-app:
    build: https://github.com/my-company/my-project.git
    ...
```

Docker containers communicate between themselves in **networks** created, implicitly or through configuration, by Docker Compose. A service can communicate with another service on the same network by simply referencing it by container name and port (for example network-example-service:80), provided that we've made the port accessible through the expose keyword:

```
services:
  network-example-service:
    image: karthequian/helloworld:latest
    expose:
      - "80"
```

In this case, by the way, **it would also work without exposing it**, because the expose directive is already in the image Dockerfile.

To reach a container from the host, the ports must be exposed declaratively through the ports keyword, which also allows us to choose if exposing the port differently in the host:

```
services:
  network-example-service:
    image: karthequian/helloworld:latest
```

```
    ports:
      - "80:80"
    ...
  my-custom-app:
    image: myapp:latest
    ports:
      - "8080:3000"
    ...
  my-custom-app-replica:
    image: myapp:latest
    ports:
      - "8081:3000"
    ...
```

Port 80 will now be visible from the host, while port 3000 of the other two containers will be available on ports 8080 and 8081 in the host. This powerful mechanism allows us to **run different containers exposing the same ports** without collisions.

Finally, we can define additional **virtual networks** to segregate our containers:

```
services:
  network-example-service:
    image: karthequian/helloworld:latest
    networks:
      - my-shared-network
    ...
  another-service-in-the-same-network:
    image: alpine:latest
    networks:
      - my-shared-network
    ...
  another-service-in-its-own-network:
    image: alpine:latest
    networks:
      - my-private-network
    ...
networks:
  my-shared-network: {}
  my-private-network: {}
```

In this example, we can see that **another-service-in-the-same-network** will be able to ping and to reach port 80 of network-example-service, while **another-service-in-its-own-network** won't.

Volumes

There are three types of volumes: **anonymous, named, and host** ones.

Docker manages both anonymous and named volumes, automatically mounting them in self-generated directories in the host. While **anonymous** volumes were useful with older versions of Docker (pre 1.9), **named** ones are the suggested way to go nowadays. Host volumes also allow us to specify an existing folder in the host.

We can configure **host** volumes at the service level and **named** volumes in the outer level of the configuration, in order to make the latter visible to other containers and not only to the one they belong:

```
services:
  volumes-example-service:
    image: alpine:latest
    volumes:
      - my-named-global-volume:/my-volumes/named-global-volume
      - /tmp:/my-volumes/host-volume
      - /home:/my-volumes/readonly-host-volume:ro
    ...
  another-volumes-example-service:
    image: alpine:latest
    volumes:
      - my-named-global-volume:/another-path/the-same-named-global-volume
    ...
volumes:
  my-named-global-volume:
```

Here, both containers will have read/write access to the **my-named-global-volume** shared folder, no matter the different paths they've mapped it to. The two host volumes, instead, will be available only to **volumes-example-service**.

The /tmp folder of the host's file system is mapped to the /my-volumes/host-volume folder of the container.

This portion of the file system is writeable, which means that the container can not only read but also write (and delete) files in the host machine.

We can mount a volume in read-only mode by appending :ro to the rule, like for the /home folder (we don't want a Docker container erasing our users by mistake).

Declaring Dependencies

Often, we need to create a dependency chain between our services, so that some services get loaded before (and unloaded after) other ones. We can achieve this result through the **depends_on** keyword:

```
services:
  kafka:
    image: wurstmeister/kafka:2.11-0.11.0.3
    depends_on:
      - zookeeper
    ...
  zookeeper:
    image: wurstmeister/zookeeper
    ...
```

Environment Variables

Working with environment variables is easy in Compose. We can define static environment variables, and also define dynamic variables with the ${} notation:

```
services:
  database:
    image: "postgres:${POSTGRES_VERSION}"
    environment:
      DB: mydb
      USER: "${USER}"
```

There are different methods to provide those values to Compose.

For example, one is setting them in a .env file in the same directory, structured like a .properties file, key=value:

```
POSTGRES_VERSION=alpine
USER=foo
```

Otherwise, we can set them in the OS before calling the command:

```
export POSTGRES_VERSION=alpine
export USER=foo
docker-compose up
```

Finally, we might find handy using a simple one-liner in the shell:

```
POSTGRES_VERSION=alpine USER=foo docker-compose up
```

Startup and Shutdown

Create and start the containers, the networks, and the volumes defined in the configuration with up:

```
docker-compose up
```

After the first time, however, we can simply use start to **start the services**:

```
docker-compose start
```

To safely stop the active services, we can use stop, which will **preserve containers, volumes, and networks**, along with every modification made to them:

```
docker-compose stop
```

To reset the status of our project, instead, we simply run down, which will **destroy everything with only the exception of external volumes**:

```
docker-compose down
```

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 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.

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.

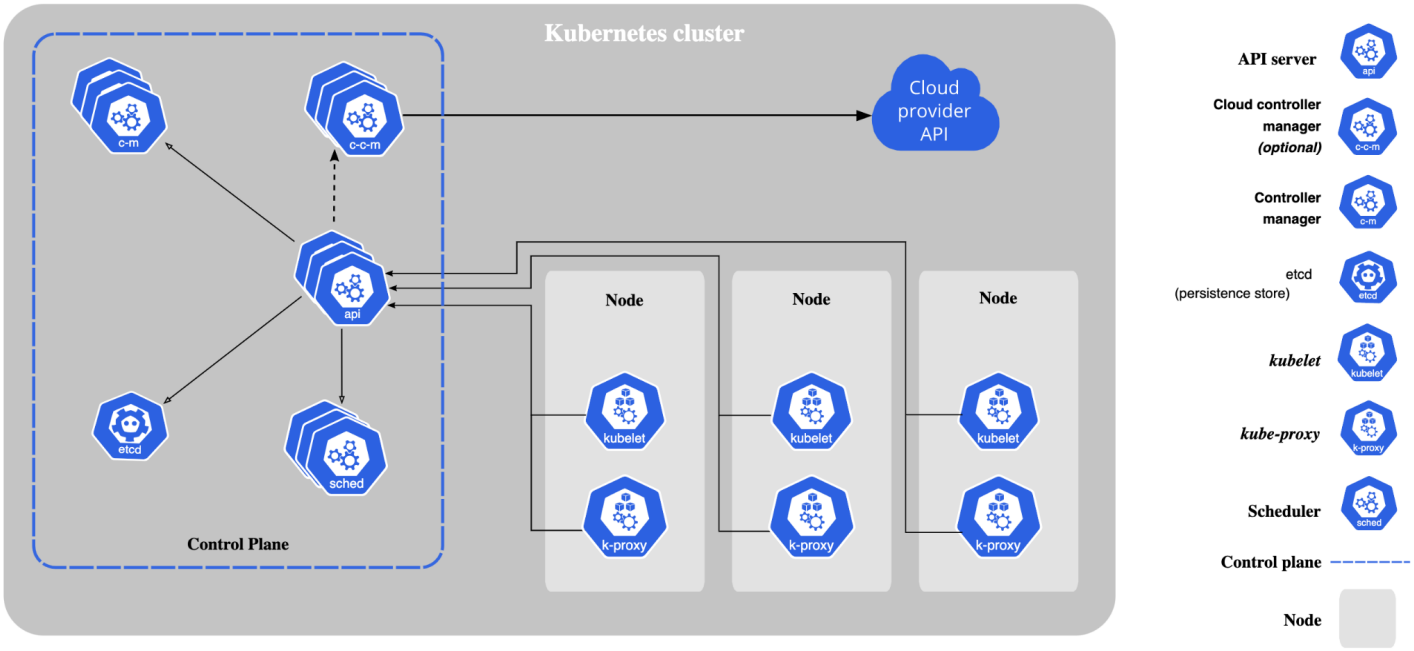
- Does not limit the types of applications supported.
- Does not deploy source code and does not build your application
- Does not provide application-level services, such as middleware (for example, message buses), data-processing frameworks (for example, Spark), databases (for example, MySQL), caches, nor cluster storage systems (for example, Ceph) as built-in services.
- Does not dictate logging, monitoring, or alerting solutions.
- 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.



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 VMs.

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 that 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.

- Endpoints controller: Populates the Endpoints object (that is, joins Services & Pods).
- Service Account & Token controllers: Create default accounts and API access tokens 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 that cloud platform from components that only 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.

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 several container runtimes: Docker, containerd, CRI-O, and any 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.

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.

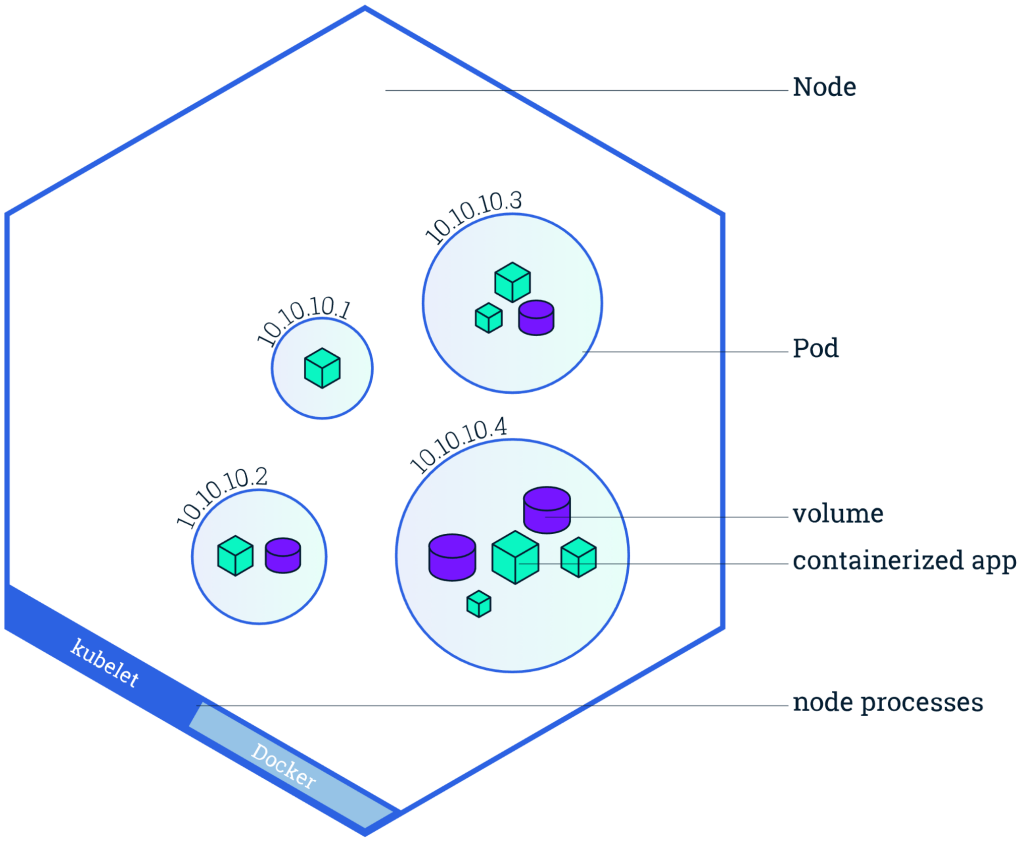
Cluster-level Logging

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

Cluster architecture

Nodes

Kubernetes runs your workload by **placing containers into Pods to run on Nodes**. A node may be a **virtual or physical machine**, depending on the cluster. Each node is managed by the control plane and contains the services necessary to run Pods.



Typically you have several nodes in a cluster; in a learning or resource-limited environment, you might have only one node.

The components on a node include the kubelet, a container runtime, and the kube-proxy.

There are two main ways to have Nodes added to the API server:

- The kubelet on a node self-registers to the control plane
- You (or another human user) manually add a Node object

```
{
  "kind": "Node",
  "apiVersion": "v1",
  "metadata": {
```

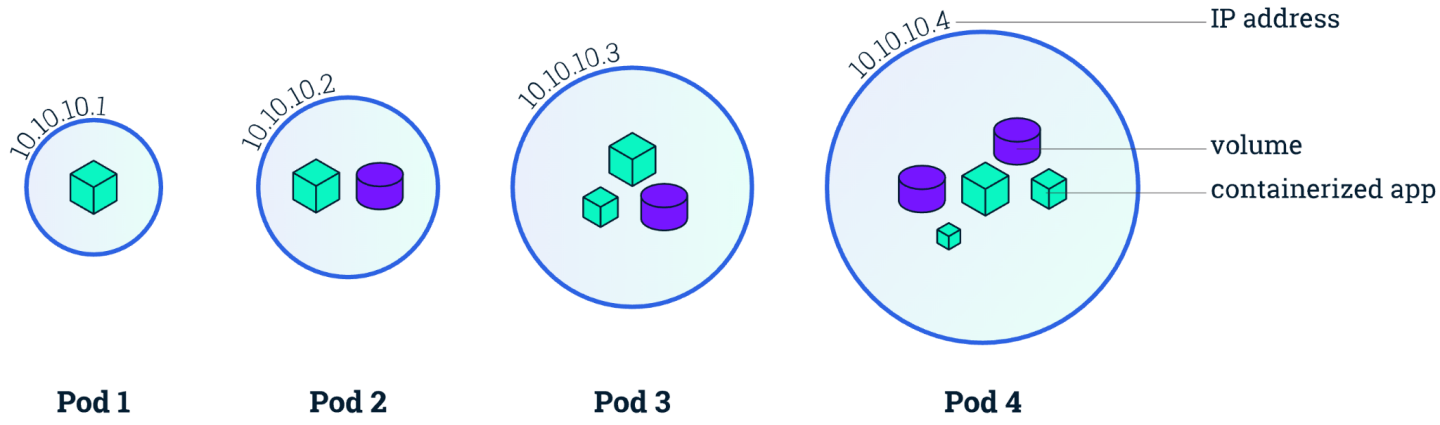
```
    "name": "10.240.79.157",
    "labels": {
      "name": "my-first-k8s-node"
    }
  }
}
```

Kubernetes creates a Node object internally (the representation). Kubernetes checks that a kubelet has registered to the API server that matches the metadata.name field of the Node. If the node is healthy (i.e. all necessary services are running), then it is eligible to run a Pod. Otherwise, that node is ignored for any cluster activity until it becomes healthy.

The name identifies a Node. Two Nodes cannot have the same name at the same time. Kubernetes also assumes that a resource with the same name is the same object. In case of a Node, it is implicitly assumed that an instance using the same name will have the same state (e.g. network settings, root disk contents). This may lead to inconsistencies if an instance was modified without changing its name. If the Node needs to be replaced or updated significantly, the existing Node object needs to be removed from API server first and re-added after the update.

Pods

A Pod is a **group of one or more application containers** (such as Docker or rkt) that includes **shared storage** (volumes), a **unique cluster IP address** and information about how to run them (like container image version or specific ports). **Containers within a Pod share an IP Address and port space**. Containers of the same Pod are always co-located and co-scheduled, and run in a shared context on the same node. A Pod models an application-specific "**logical host**" and contains one or more application containers which are relatively tightly coupled. An example of container that would fit on the same Pod with our the NodeJS app, would be a side container that feeds the data published by the webserver. In a pre-container world, they would have run on the same physical or virtual machine.



Pods are tied to the Node where they are deployed and remain there until termination (according to restart policy) or deletion. In case of a Node failure, new identical Pods will be deployed on other available Nodes. **The Pod is the atomic deployment unit on the Kubernetes platform**. When we trigger a Deployment on Kubernetes, it will create Pods with containers inside them, not containers directly.

Benefits of a pod: when pods contain multiple containers, communications, and data sharing between them is simplified. Since all containers in a pod **share the same network namespace**, they can locate each other and communicate via localhost. Pods can communicate with each other by using another pods IP address or by referencing a resource that resides in another pod.

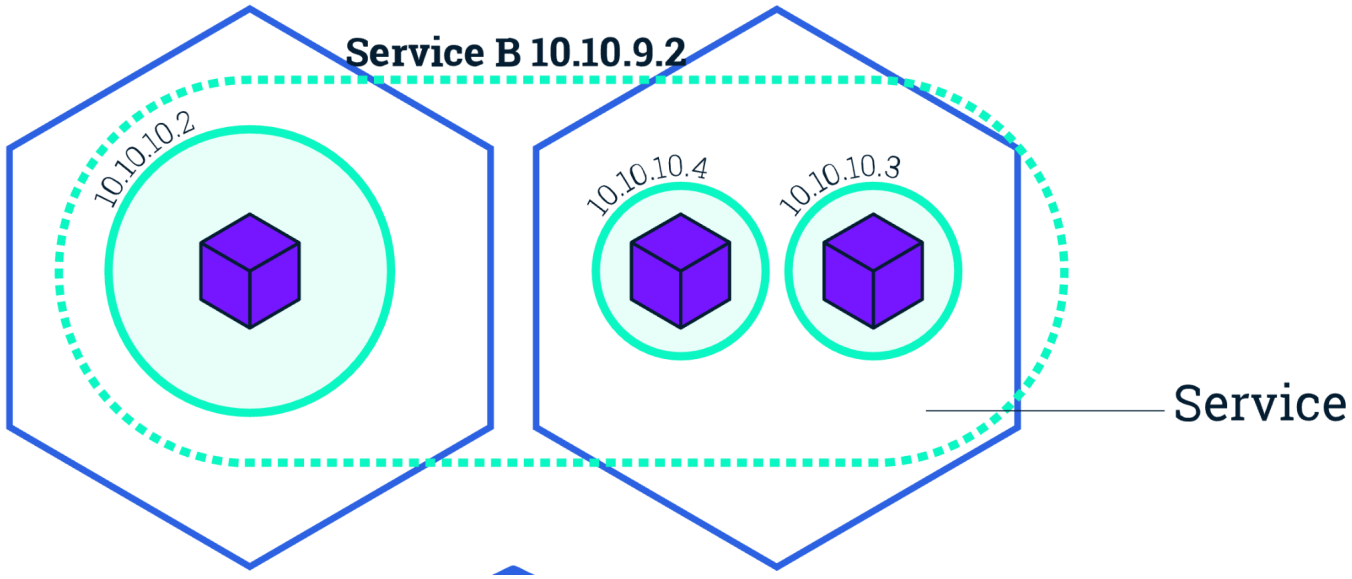
Pods can include containers that run when the pod is started, say to perform initiation required before the application containers run. Additionally, pods simplify scalability, enabling replica pods to be created and shut down automatically based on changes in demand.

Services

An abstract way to expose an application running on a set of Pods as a network service.

While Pods do have their own unique IP across the cluster, those IP's **are not exposed outside Kubernetes**. Taking into account that over time Pods may be terminated, deleted or replaced by other Pods, we need a way to let other Pods and applications automatically **discover each other**. Kubernetes addresses this by grouping Pods in Services. A Kubernetes Service is an abstraction layer which defines a **logical set of Pods** and enables **external traffic exposure, load balancing and service discovery** for those Pods.

With Kubernetes you don't need to modify your application to use an unfamiliar service discovery mechanism. Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them.



This abstraction will allow us to expose Pods to traffic originating from outside the cluster. Services have their own unique cluster-private IP address and expose a port to receive traffic. If you choose to expose the service outside the cluster, the options are:

- **LoadBalancer** - provides a public IP address (what you would typically use when you run Kubernetes on GKE or AWS)
- **NodePort** - exposes the Service on the same port on each Node of the cluster using NAT (available on all Kubernetes clusters, and in Minikube)

A Service in Kubernetes is a REST object, similar to a Pod:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
```

```
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

This specification creates a new Service object named "my-service", which targets TCP port 9376 on any Pod with the app=MyApp label.