Kubernetes 101 Series

Concepts & Building Blocks

Part 1

MAGALIX

Contents

To be Continued

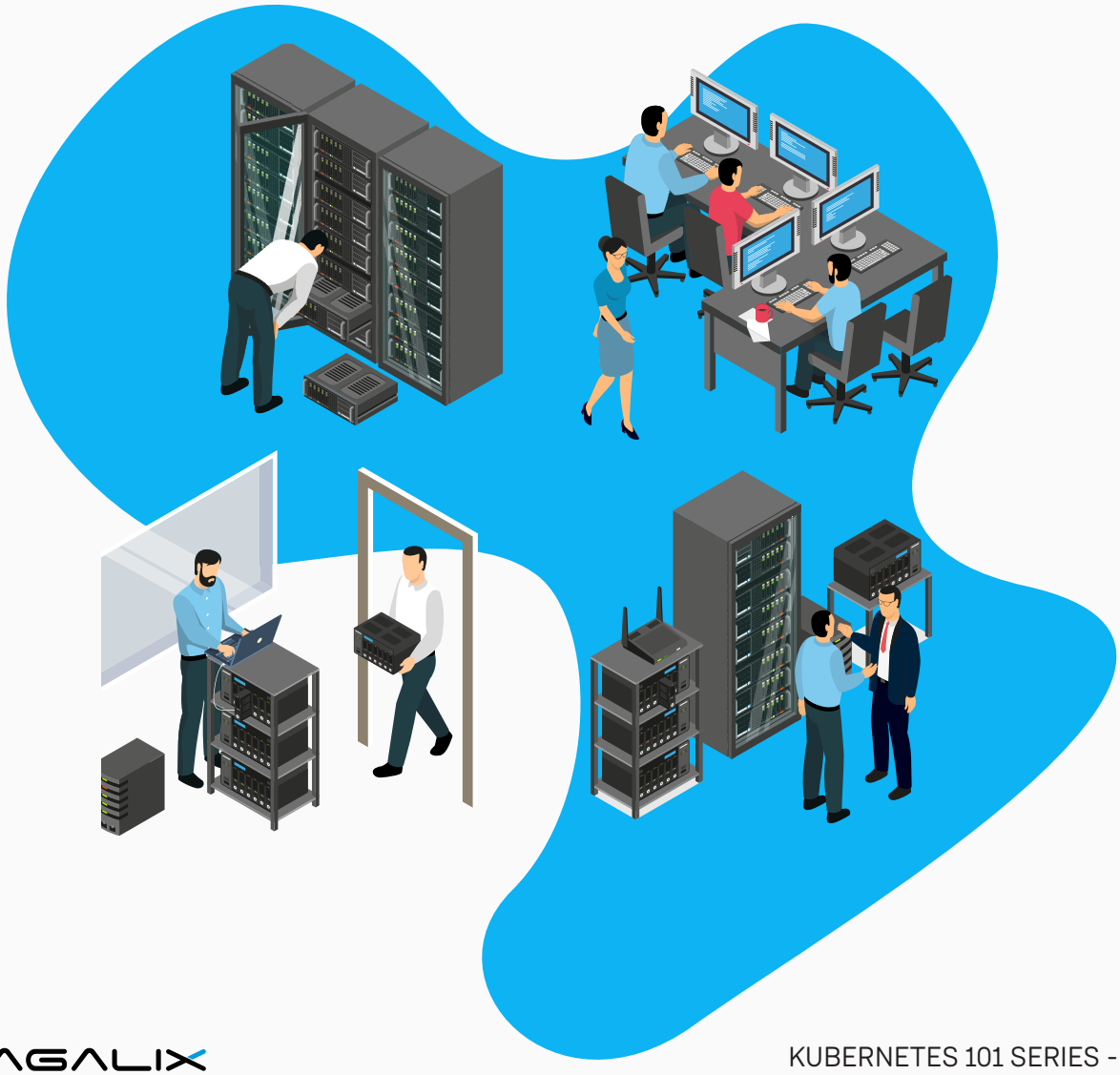
Part2 - Configs, Secrets & Observability

Part3 - Networking, Resources Management & Security

Introduction

Less than tweeny years ago, if you wanted to have an infrastructure to host your application, you needed the following:

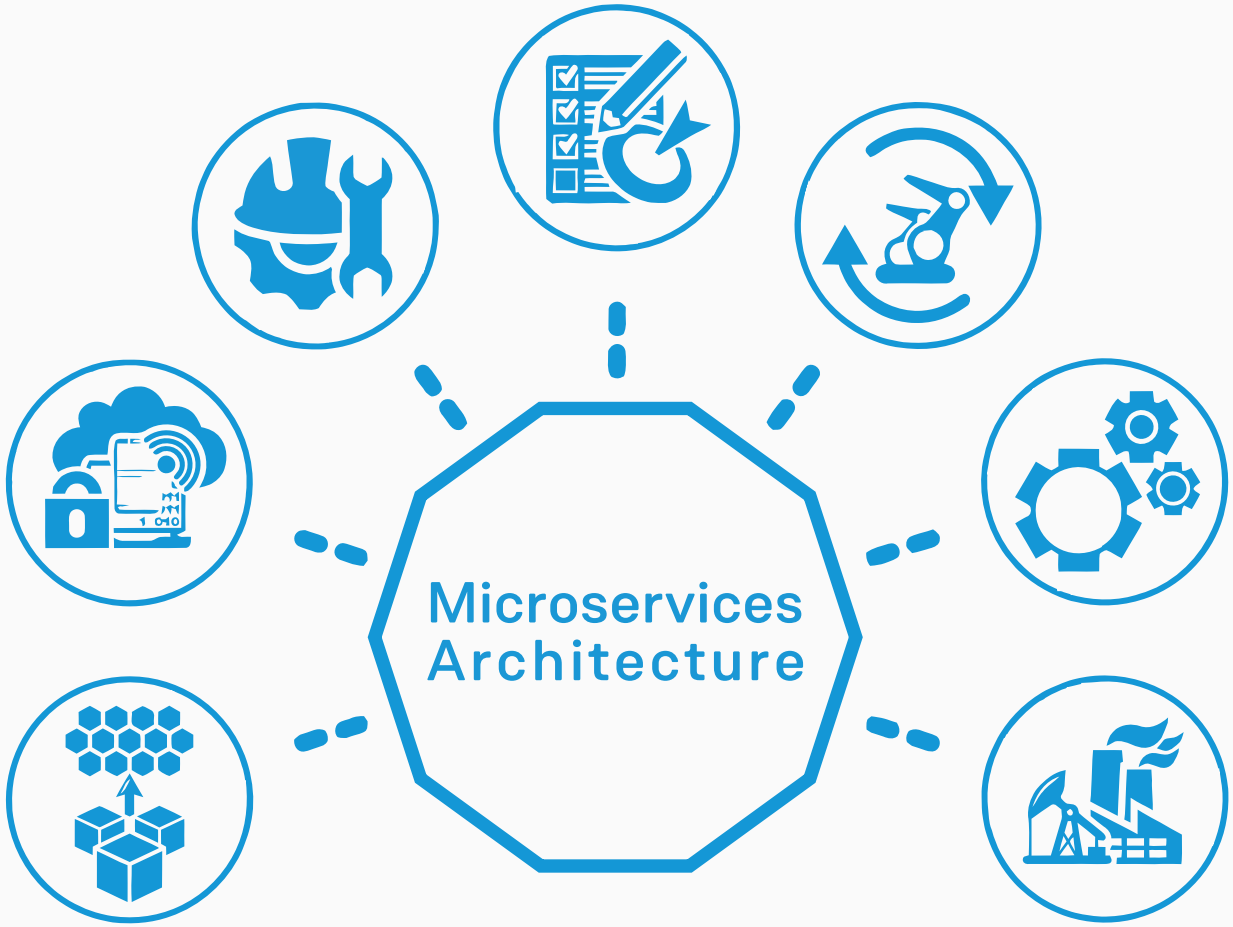
- A room with a special setup to meet data center requirements (raised floor, air-condition, several power-supply sources, etc.)
- *At least* one rack hosting the server.
- The required networking hardware (routers, switches, firewall...etc)
- A team of engineers that are tasked with managing this infrastructure for you. Activities like hardware deployment, OS installation, network administration among others are some of the tasks that the operators are responsible for.



In the computer world, twenty calendar years are equivalent to several decades. The above describes the prehistoric ages of software deployment. Today, this whole setup was replaced by cloud providers; those giants, complex companies that decided to rent parts of their infrastructure to regular customers like you and me. Payment is by the hour of usage and you can pay for the amount of computing power, storage, and network bandwidth you are using. Of course, you're already aware that we are describing what's referred to as "cloud computing". But what does this have to do with Kubernetes? **Read on, please.**

Since infrastructure has become so elastic that you can literally spin out a complete setup that contains one or more virtual machines, with their underlying storage, network, firewall rules, and backend database in less than a minute. That ease of creating/destroying allowed for breaking monolithic applications into smaller, interdependent components that communicate with each other through a well-defined protocol, often HTTP. The advantages of this setup are numerous, for example:

- You can have separate teams each working on their own service. This way, each team can work independently without interfering with other teams.
- When the application is running, it serves content from multiple, combined services. If one of them goes down, the rest of the application is not necessarily affected. For example, if part of your application displays the latest posts in a ticker at the side of the screen, then if this posts service is down, you can just display a nice "will be back" message in its place. The rest of the application is functioning normally and you're not losing your clients.
- Combing with cloud technologies, you can make intelligent, cost-saving decisions that weren't possible in the monolithic era. For example, if your application has a number of services that are CPU intensive, you can choose to host them on machines with multiple cores, fast CPUs. the rest of the application can be hosted on less expensive machines.
- You can use different programming languages depending on your needs. For example, perhaps you want to introduce new features of your application that use AI. You know that Python is among the most popular languages in machine learning and AI. Since you are using a separate service for each component of the application, you can introduce the new addition as a service that is written in Python. The service exposes itself to the rest of the application through a REST API (no pun intended!).



The above setup is commonly referred to as the Microservices Architecture.

The microservices architecture became very popular with the advent of cloud computing. Software giants started implementing it like Netflix, Uber, Sound Cloud among many others.

Now, microservices may be a great way of architecting your application. It provides many benefits, but it also has its own drawbacks, for example:

- Since we are using multiple services, there should be some sort of high availability among them. So, every service should have at least one replica so that if one dies, the other is still working and serving content.
- There must be a way to detect when a service dies so that it can either get restarted or moved from one server (also called node) to the other if the cause of the crash is the whole node went down.

- Since we have multiple services, each is served by at least two replicas for high availability, there should be a better way of deploying updates without having to bring the whole application down.

So far we've been discussing cloud computing and microservices from a very high level. We haven't touched any implementation details yet. The reason is that the implementation is always left to the user to decide. Having said that, it's worth noting that most people have resorted to using Docker containers for microservices. A container is a lightweight, self-contained environment that can be deployed to the same host along with other containers. Each service can be represented by a container. But with so many containers, nodes, and services, it becomes very hard (or next to impossible) to manage the environment by a human. Imagine the neverending tasks that have to be done on a 24/7 basis. For example, which containers went down, how many times they were restarted, is the node healthy? Does it have enough CPU and memory to host another container? And so on. For that reason, there had to be a container orchestrator system.

There are many tools on the market that serve as good container orchestration systems. For example, Docker Swarm, Apache Mesos, and others. However, the most well-known technology that took the orchestration world by force is Kubernetes. Originally founded by Google and open-sourced in 2014, Kubernetes proved to be one of the most reliable systems for managing containers at scale. The technology is currently used in many places both in testing as well as in production environments and it's doing a great job.

In this book, we are giving you a gentle introduction to Kubernetes. We start by defining the technology and how it works, then we delve deeper into the inner details that make this sophisticated system work in such harmony. We study Pods, Services, ReplicaSets, Deployments, StatefulSet, and Jobs. By the end of this book, you should have a working knowledge of Kubernetes and you should be able to build your own cluster.

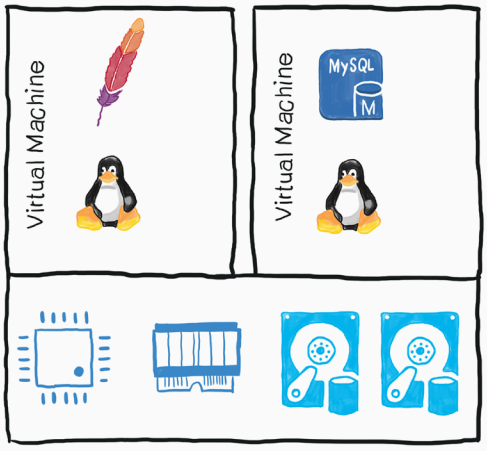
1. Kubernetes Concpets & Why It Matters

If you work in DevOps, or the IT field in general, you've surely heard the term Kubernetes. In this chapter, we explore Kubernetes from a 10,000-foot overview. We'll also shed light on some of its most important use cases and best practices.

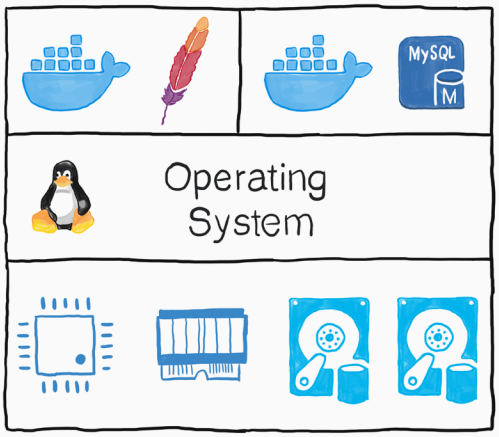
To fully understand the technology, you need to be aware of the "why and how" that brought Kubernetes, a container orchestration tool, into existence. The Kubernetes story starts with containers - to appreciate the merits of containers, let's see how software deployment mechanisms evolved over time

1.1 Docker Containers Changed How We Deploy Software

In the old days, software deployment was hard, time-consuming, and error-prone. To install an application, you needed to purchase a number of physical machines and pay for extra CPU power and more memory than you might actually need. A few years later, virtualization was dominant - this saved you some cost since one powerful bare-metal server can host multiple machines. Thus, CPU and memory could be shared. Now, machines can be split into even smaller parts than virtual servers: ***containers***. Containers became extremely popular just a few years ago. So, what exactly is a Linux container? And where does Docker fit?



Software deployment in the old days



software deployment today

A container provides a type of virtualization just like virtual machines. However, while a hypervisor provides a hardware isolation level, containers offer process isolation levels. To understand this difference, let's look at an example.

Instead of creating a virtual machine for Apache and another for MySQL, you decide to use containers. Now, your stack looks like the illustration below

A container is nothing but a set of processes on the operating system. A container works in complete isolation from other processes/containers through Linux kernel features, such as cgroups, chroot, UnionFS, and namespaces.

This means you'll only pay for one physical host and install one OS, and then you can run as many containers as your hardware can handle. Overall, this will boost performance and reduce the number of operating systems that you need to run on the same host - costing you less storage, memory, and CPU power.

In 2010, Docker was founded (Docker refers to both the company and the product). Docker made it very easy for users and companies to utilize containers for software deployment. However, it's important to note that Docker is not the only tool in the market that does this. Other applications exist like rkt, Apache Mesos, and LXC, to name a few. Docker just happens to be the most popular via innovation and utilit.

1.2 Containers And Microservices: The Need For An Orchestrator

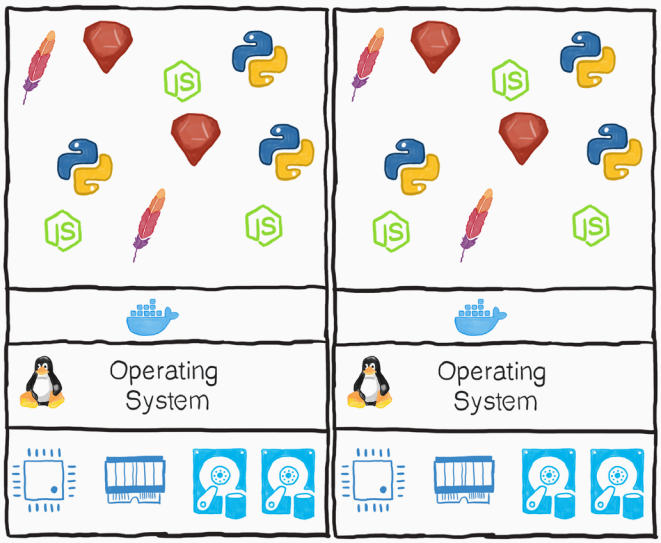
The ability to run complete services in the form of processes (a.k.a containers) on the same OS was revolutionary. It brought a lot of possibilities of its own:

- Because containers are way cheaper and faster than virtual machines, large applications could now be broken down into small, interdependent components, each running in its own container. This architecture became known as *microservices*.
- With the microservices architecture becoming more dominant, applications had more freedom to get larger and richer. Previously, a monolithic application grew till a certain limit where it became cumbersome, harder to debug, and very difficult to be re-deployed. However, with the advent of containers, all that you need to do to add more features to an application is to build more containers/services. With IaC (Infrastructure as Code), deployment is as easy as running a command against a configuration file.
- Today, it is no longer acceptable to have a downtime. The user simply does not care if your application is experiencing a network outage or your cluster nodes crashed. If your system is not running, the user will simply switch to your competitor.

All of the above encourages IT professionals to do one thing: create as many containers as possible. However, this also has drawbacks:

- - Containers are processes, and processes are ephemeral by nature. What happens if a container crashes?
- - To achieve high availability, you create more than one container for each component. For example, two containers for Apache, each hosting a web server. But, which one of them will respond to client requests?
- - When you need to update your application, you want to make use of having multiple containers for each service. You will deploy the new code on a portion of the containers, recreate them, then do the same on the rest. But, it's very hard to do this manually - not to mention, it's error-prone.

For instance, let's say you have a microservices application that has multiple services running Apache, Ruby, Python, and NodeJS. You can use containers to make the best use of the hardware at hand. However, with so many containers dispersed on your nodes without being managed, your infrastructure may look as follows:



Node 01 Node 02

Sir, you need a container orchestrator.

1.3 Please Welcome Kubernetes

Kubernetes is a container orchestration tool. Orchestration is another word for lifecycle management. A container orchestrator does many tasks, including:

- Container provisioning.
- Maintaining the state (and number) of running containers.
- Distribute application load evenly on the hardware nodes by moving containers from one node to the other.
- Load balancing among containers that host the same service.
- Handling container persistent storage.
- Ensuring that the application is always available even when rolling out updates.

Just like Docker not being the only container platform out there, Kubernetes is not the sole orchestration tool on the market. There are other tools like Docker Swarm, Apache Mesos, Marathon, and others. So, what makes Kubernetes the most widely used?

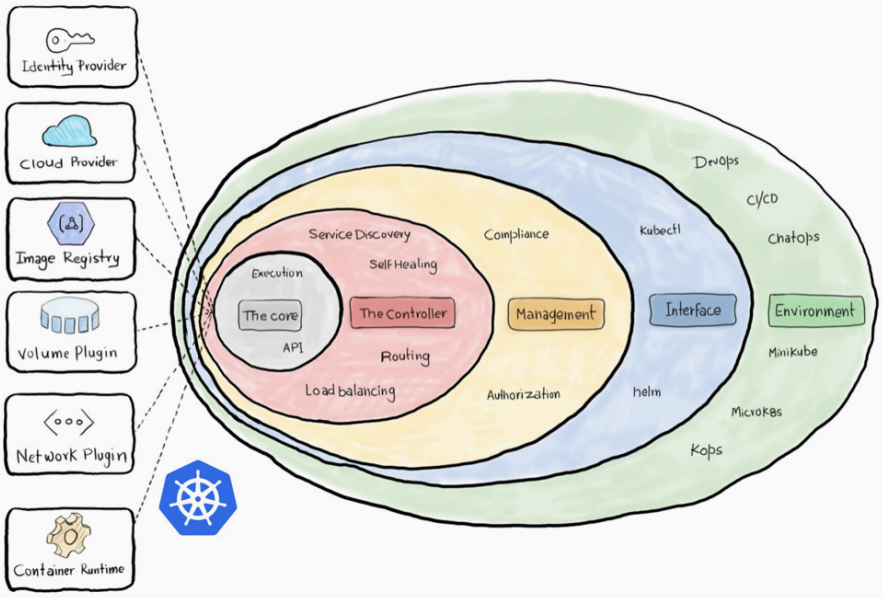
1.4 Why Is Kubernetes So Popular?

Kubernetes was originally developed by the software and search giant, Google. It was a branch of their **Borg** project. Since its inception, Kubernetes received a lot of momentum from the open-source community. It is the main project of the Cloud Native Computing Foundation. Some of the biggest market players are backing it: Google, AWS, Azure, IBM, and Cisco to name a few.

1.5 Kubernetes Architecture And Its Environment?

Kubernetes is the Greek word for helmsman or captain. It is the governor of your cluster, the maestro of the orchestra. To be able to do this critical job, Kubernetes was designed in a highly modular manner. Each part of the technology provides the necessary foundation for the services that depend on it. The illustration below represents a high overview of how the application works. Each module is contained inside a larger one that it relies upon to function. Next, let's dig deeper into each one.

An overview of the landscape of Kubernetes as a system:



Kubernetes Core Features

Also referred to as the control plane, it's the most basic part of the whole system. It offers a number of RESTful APIs that enable the cluster to do its most basic operations. The other part of the core is execution. Execution involves a number of controllers like replication controller, replicaset, deployments. It also includes the **kubelet**, which is the module responsible for communicating with the container runtime.

The core is also responsible for contacting other layers (through kubelet) to fully manage containers. Let's have a brief look at each of them:

Container Runtime

Kubernetes uses **Container Runtime Interface (CRI)** to transparently manage your containers without necessarily having to know (or deal with) the runtime used. When we discussed containers, we mentioned that Docker, despite its popularity, is not the only container management system available. Kubernetes

uses **containerd** (pronounced container d) by default as a container runtime. This is how you're able to issue standard Docker commands against Kubernetes containers. It also uses rkt as an alternative runtime. Don't be too confused about this part - these are the very innermost workings of Kubernetes that, although you need to understand, you probably won't have to deal with. Kubernetes abstracts this layer through its rich set of APIs.

The Network Plugin

As we discussed earlier, a container orchestration system is responsible (among other things) for managing the network nexus where containers and services communicate. Kubernetes uses a library called **Container Network Interface (CNI)** as a medium between the cluster and various network providers. There are a number of network providers that can be used in Kubernetes - this number is constantly changing. To name a few:

- Weave net
- Contiv
- Flannel
- Calico

You might be asking: why does Kubernetes need more than one networking provider? Kubernetes was designed mainly to be deployed in diverse environments. A Kubernetes node can be anything from a bare-metal physical server, a virtual machine, or a cloud instance. With such diversity, you have a virtually endless number of options for how your containers will communicate with each other. That is why Kubernetes designers choose to abstract the network provider layer behind CNIt.

The Volume Plugin

A volume broadly refers to the storage that will be available for the pod. A pod is one or more containers managed by Kubernetes as a single unit. Because Kubernetes was designed to be deployed in multiple environments, there is a level of abstraction between the cluster and the underlying storage.

Kubernetes also uses the CSI (Container Storage Interface) to interact with various storage plugins that are already available.

Image Registry

Kubernetes must contact an image registry (whether public or private) to be able to pull images and spin out containers.

Cloud Provider

Kubernetes can be deployed on almost any platform. However, the majority of users resort to cloud providers like AWS, Azure, or GCP to save cost. Kubernetes depends on the cloud provider APIs to perform scalability and resource provisioning tasks, such as provisioning load balancers, accessing cloud storage, utilizing the inter-node network, etc.

Identity Provider

If you're provisioning a Kubernetes cluster in a small company with a small number of users, authentication won't be a big issue. You can create an account for each user and that's it. But, if you're working in a large enterprise, with hundreds or even thousands of developers, operators, testers, security professionals, etc., then having to manually create an account for each person may quickly turn into a nightmare. Kubernetes designers had that in mind when working on the authentication mechanism. You can use your own identity provider system to authenticate your users to the cluster as long as it uses OpenID connect.

1.6 Kubernetes Controllers Layer

This is also referred to as the service fabric layer. It is responsible for some higher-level functions of the cluster: routing, self-healing, load balancing, service discovery, and basic deployment (for more info, https://kubernetes.io/docs/concepts/services-networking/, and https://kubernetes.io/docs/concepts/workloads/controllers/deployment/), among other things.

Management Layer

This is where policy enforcement options are applied. In this layer, functions like metrics collection, and autoscaling are performed. It also controls authorization, and quotas among different resources like the network and storage. You can learn more about resource quotas here.

The Interface Layer

In this layer, we have the client-facing tools that are used to interact with the cluster. **kubectl** is the most popular client-side program out there. Behind the scenes, it issues RESTful API requests to Kubernetes and displays the response either in JSON or YAML depending on the options provided. kubectl can be easily integrated with other higher-level tools to facilitate cluster management.

In the same area, we have **helm**, which can be thought of as an application package manager running on top of Kubernetes. Using **helm-charts**, you can build a full application on Kubernetes by just defining its properties in a configuration file.

1.7 DevOps And Infrastructure Environment

Kubernetes is one of the busiest open-source projects in use. It has a large, vibrant community and it's constantly changing to adapt to new requirements and challenges. Kubernetes provides a tremendous number of features. Although it is only a few years old, it is able to support almost all environments. Kubernetes is used in many leading software building/deployment practices including:

- **DevOps**: provisioning ephemeral environments for testing and QA is easier and faster.
- **CI/CD:** building continuous integration/deployment, and even delivery pipelines, are more seamless endeavors using Kubernetes-managed containers. You can easily integrate tools like Jenkins, TravisCI, Drone CI with the Kubernetes cluster to build/test/deploy your applications and other cloud components.
- **ChatOps**: chat applications like Slack can easily be integrated with the rich API

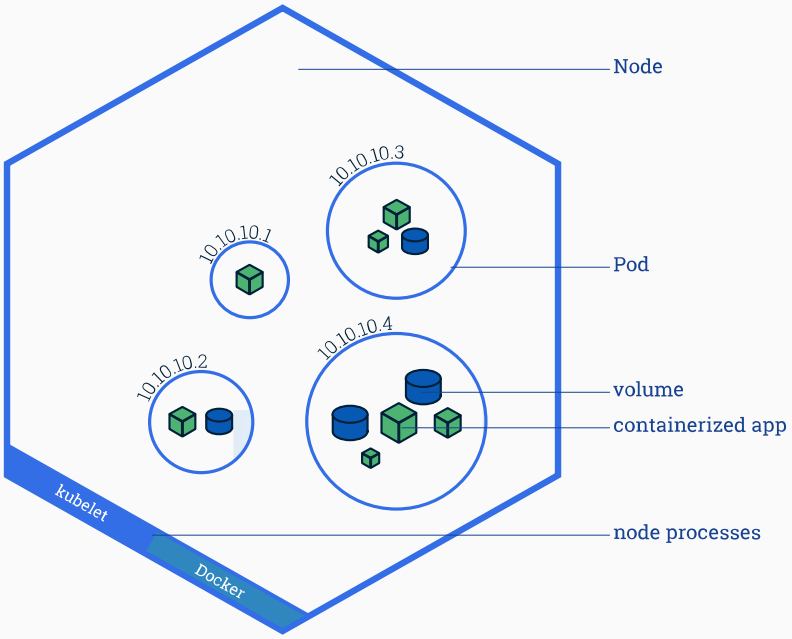
set provided by Kubernetes to monitor, and even manage, the cluster.

- **Cloud-managed Kubernetes**: most cloud providers offer products that already have Kubernetes installed. For example AAWS EKS, Google GKE, and Azure AKS.
- **GitOps**: everything in Kubernetes is managed through code (YAML files). Using version control systems like Git, you can easily manage your cluster through pull requests - you don't even have to use kubectl.

Summary

In this chapter, we looked at a 10,000-foot overview of Kubernetes. We briefly covered the concept of containers, why they're so popular, and the difference between a container and a virtual machine. Finally, we discussed Kubernetes as a tool, why it came into existence, and how it works at a very basic level. We intentionally avoided any Kubernetes-specific lingo (as much as possible) so that we could focus on the core concepts. In future chapters, we'll delve deeper into the ideas that we touched here, and explain how they work under the hood. Kubernetes is a very large topic and you can easily get lost going deeper into any one of its components. If that happens, you can always return to this chapter to review the basics.

2. Pods, The Cluster Sailors



If Kubernetes is the helmsman, and Docker is the dock worker, then pods are the sailors of the ship (or cluster).

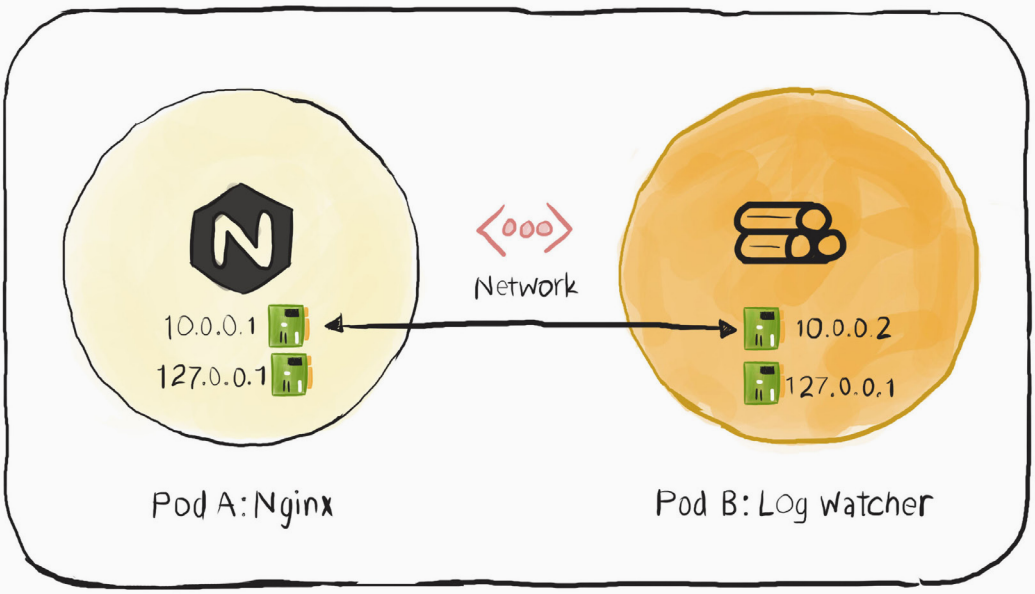
2.1 What Is A Pod in Kubernetes?

A pod is the smallest unit in a Kubernetes cluster. A pod may contain one or more containers. Pods can be scheduled (in Kubernetes terminology, scheduling means deploying) to a node. When you first learn about pods, you may think of them as containers, yet they aren't. As a container, a pod is a self-contained logical process, with an isolated environment. It has its own IP address, storage, hostname, etc. However, a pod can host more than one container. So, a pod can be thought of as a container of *containers*.

2.2 When, And Why Would We, Host More Than One Container In A Pod?

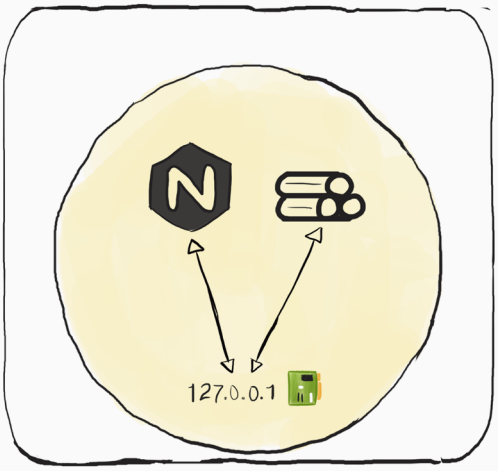
In the microservices architecture, each module should live in its own space and communicate with other modules following a set of rules. But, sometimes we need to deviate a little from this principle.

Suppose that you have an Nginx web server running (see below illustration). We need to analyze Nginx logs in real-time. The logs we need to parse are obtained from GET requests to the web server. The developers created a log watcher application that will do this job, and they built a container for it. Under typical conditions, you'd have a pod for Nginx and another for the log watcher. However, we need to eliminate any network latency so that the watcher can analyze logs the moment they are available. A solution for this is to place both containers on the same pod.



.

Having both containers on the same pod allows them to communicate through the loopback interface as if they were two processes running on the same host. They also share the same storage volume.



2.3 Our First Pod

In order to work with this example, you'll need a Kubernetes cluster running. Open your favorite text editor, create a new file called pods01.yaml and add the following to it:

```
apiVersion: v1
kind: Pod
metadata:
  name: webserver
spec:
  containers:
  - name: webserver
      image: nginx:latest
      ports:
      - containerPort: 80
```

Let's first have a quick look at each line in this file:

- **apiVersion**: this is the version of the API used by the cluster. With new versions of Kubernetes being released, new functionality is introduced and, hence, new API versions may be defined. For the pod object, we use API version v1.
- **Metadata**: here we can define data about the object we are about to create. In this example, we only provide the name of the pod. But you can provide other details like the namespace.
- The **spec** part defines the characteristics that a given Kubernetes object should have. It is the cluster's responsibility to update the status of the object to always match the desired configuration. In our example, the spec instructs that this object (the pod) should have one container with some attributes.
- **containers**: the containers part is an array where one or more container specs can be added. For example
  - The name that this container will have.
  - The image on which it is based.
  - The port(s) that will be open.

Let's *apply* this configuration to the cluster. Issue the following command:

```
kubectl apply -f pods01.yaml
```

You should see an output similar to the following:

```
pod/webserver created
```

2.4 Kubectl

Kubectl is the client tool that is used to send API requests to Kubernetes. You could've used kubectl to achieve the same results using a command like the following:

```
kubectl run webserver --image=nginx --port=80
```

However, it is highly recommended that you use YAML (or JSON) files to build and configure your cluster - this will allow you to put your infrastructure under a version control system (like Git).

Kubectl can be used for building, deleting, viewing, and updating Kubernetes resources. In fact, kubectl sends API requests to the cluster behind the scenes. So, technically you could get rid of the tool and issue all your cluster commands through an HTTP client like CURL. However, it is strongly recommended that you use kubectl; it's far easier than sending raw HTTP requests.

2.5 Viewing Your Pods

One of the most repetitive tasks you'll do as a K8s (K8s is the de facto abbreviation for Kubernetes) administrator is to view the status of your cluster resources. The following command allows you to view the pods running on your cluster:

```
kubectl get pods
```

The output should be similar to the following:

```
NAME        READY   STATUS   RESTARTS   AGE
webserver   1/1     Running  0          19m
```

- Notice that the READY column is displaying 1/1, which means that this pod has only one container (out of one), and it's ready.

- The STATUS is running, which means that the pod is in service and it can receive and respond to requests.
- The RESTARTS column represents how many times this pod was restarted. Kubernetes will attempt to restart a failing pod whenever it fails.
- The AGE represents how much time has passed since this pod was created.

2.6 Which Node Is This Pod Running On?

You'll typically run Kubernetes on more than one node. The master node is responsible for scheduling pods on nodes. Kubernetes allows you to force a pod to be scheduled on a specific node. Let's say you have recently purchased a powerful machine with a huge amount of memory, and you want your MongoDB pod(s) to always use this node in order to benefit from the increased RAM. But, is the pod running on this node as you had hoped? You can use the following command to make sure it is:

```
kubectl get pods -o wide
```

The output should resemble the following:

```
NAME        READY   STATUS   RESTARTS   AGE   IP          NODE
webserver   1/1     Running  1          1h    10.1.0.4    docker-for-desktop
```

Now, we have the node name (we're using Kubernetes on macOS), and we also have the internal IP address of the pod.

2.7 I Need The Output In JSON

Like other Kubernetes components, kubectl was designed to be modular. You can have its output *piped* to other tools. Commonly, tools use JSON as their preferred communication language. Kubectl allows you to have the output of its subcommands in either YAML or JSON, with YAML being the default. Let's check out a quick example:

```
kubectl get pods -o json
```

The output is way too long to be presented here - but, you can easily pipe it to a JSON parser like jq:

```
kubectl get pods -o json | jq
```

We hope by now you've come to appreciate the raw power of kubectl.

2.8 The Pod Is Failing But I Don't Know Why

While the `get pods` subcommand may be useful most of the time, there's a good chance that you'll need more details about a pod. Let's say that you ran `kubectl` get pods and the output looked like this

```
NAME        READY   STATUS         RESTARTS   AGE
webserver   0/1     ErrImagePull     0         42m
```

Later on, the output is something like:

```
NAME        READY   STATUS              RESTARTS   AGE
webserver   0/1     CrashLoopBackOff    0          45m
```

The `get pods` subcommand shows that the pod is not ready. But, that's it. To know what made the pod fail, let's issue the following command:

```
kubectl describe pods webserver
```

The output will be kind of verbose, so we're not going to show all of it here, only the important parts:

```
Normal   Pulled                 47m                    kubelet, docker-
for-desktop  Successfully pulled image "nginx:latest"
  Warning  Failed                 3m29s (x4 over 4m58s)  kubelet,
docker-for-desktop  Failed to pull image "ngnx:latest": rpc error: code
= Unknown desc = Error response from daemon: pull access denied for
ngnx, repository does not exist or may require 'docker login'
```

llt's clear now that the pod was trying to pull an image that was misspelled (`ngnx` instead of the correct `nginx`).

`kubectl describe pods` will give you more than just the status of the image. It will print any logs that the pod produces, which is invaluable when troubleshooting a misbehaving pod.

2.9 Executing Commands Against Pods

If you're a Docker user, you've probably needed to execute commands against running containers, mostly for troubleshooting purposes. A Docker exec command may look like this:

```
docker exec -it webserver /bin/bash
```

The above command will immediately execute /bin/bash against the container, which - if bash is installed - will open a shell inside the container.

Kubectl has borrowed this functionality from Docker. You can issue a very similar command against the pod as follows:

```
kubectl exec -it webserver -- /bin/bash
```

Notice that we used the pod name, not the container. Remember, pods host containers. We also used the -- to inform kubectl that we're no longer adding any more subcommands or arguments and that the forthcoming syntax is the actual command that needs to be executed against the pod.

We mentioned before that a pod can contain multiple containers, so which container will kubectl execute bash against? Good question. Kubectl will select the first available container in the pod. If you want to run the command against a specific container in a multi-container pod, you can do so by adding the -c container name option. More on that to follow later.

2.10 Terminating A Pod

By default, Kubernetes will restart a pod upon exit. Let's see this in action:

```
$ kubectl exec -it webserver -- /bin/bash -c "kill 1"
$ kubectl get pods
NAME       READY   STATUS   RESTARTS   AGE
webserver   0/1  Completed   2         3h
```

We killed Nginx, the main process that keeps the container running (hence, it has the pid of 1). When we check on the pod's status we see that the status is no longer running. But, after a few seconds, we can see that the pod is in the running state again. The AGE column is still reflecting 3 hours because Kubernetes did not recreate the pod - it only restarted it.

```
$ kubectl get pods
NAME       READY   STATUS   RESTARTS   AGE
webserver   1/1    Running   3         3h
```

But what if we really need the pod to go down? The following command will do the job:

```
kubectl delete pods webserver
```

A Kubernetes object configuration file can also be used to terminate an object just like it was used to create it:

```
kubectl delete -f pods01.yaml
```

Notice that here we didn't specify pods as the type of resource that Kubernetes will act upon - this has been intelligently inferred by kubectl from the YAML file.

Both commands may have similar results. However, a YAML file can contain more than one resource. For example, you can specify a deployment, a secret, and a volume in the same YAML file. Calling kubectl against the file will automatically create, or terminate, all objects contained in it

2.11 Now Let's Add A Second Container To The Pod

As a bonus, let's demonstrate how a pod can host more than one container. Let's create another YAML file. Call it `pods02.yaml` and add the following:

```
apiVersion: v1
kind: Pod
metadata:
  name: webserver
spec:
  containers:
  - name: webserver
      image: nginx:latest
      ports:
      - containerPort: 80
  - name: webwatcher
      image: afakharany/watcher:latest
```

The only change we made here is adding a new container to the containers array. The name of the container is now more important than before; as it will be used whenever you need to execute commands against this specific container in the pod.

Again, let's apply the configuration by issuing `kubectl apply -f pods02.yaml` then kubectl get pods. You should see an output close to the following:

```
NAME        READY   STATUS   RESTARTS   AGE
webserver   2/2     Running   0         8s
```

Notice that the READY column now has two containers running out of two.
The second container features a parser that will count the number of 'n' characters in the HTML returned from the web-server's home page. In a real-world scenario, this may be processing JSON output from an API, but let's keep things simple. The Dockerfile used to create this second container is as follows

```
FROM python:3.6
RUN pip install requests
COPY watcher.py /
ENTRYPOINT ["python", "/watcher.py"]
```

And the watcher.py file contents are:

```
import requests
import time
r = requests.get("http://127.0.0.1").text
while True:
    num = r.count("n")
    print("There are " + str(num) + " occurrences of 'n'")
    time.sleep(5)
```

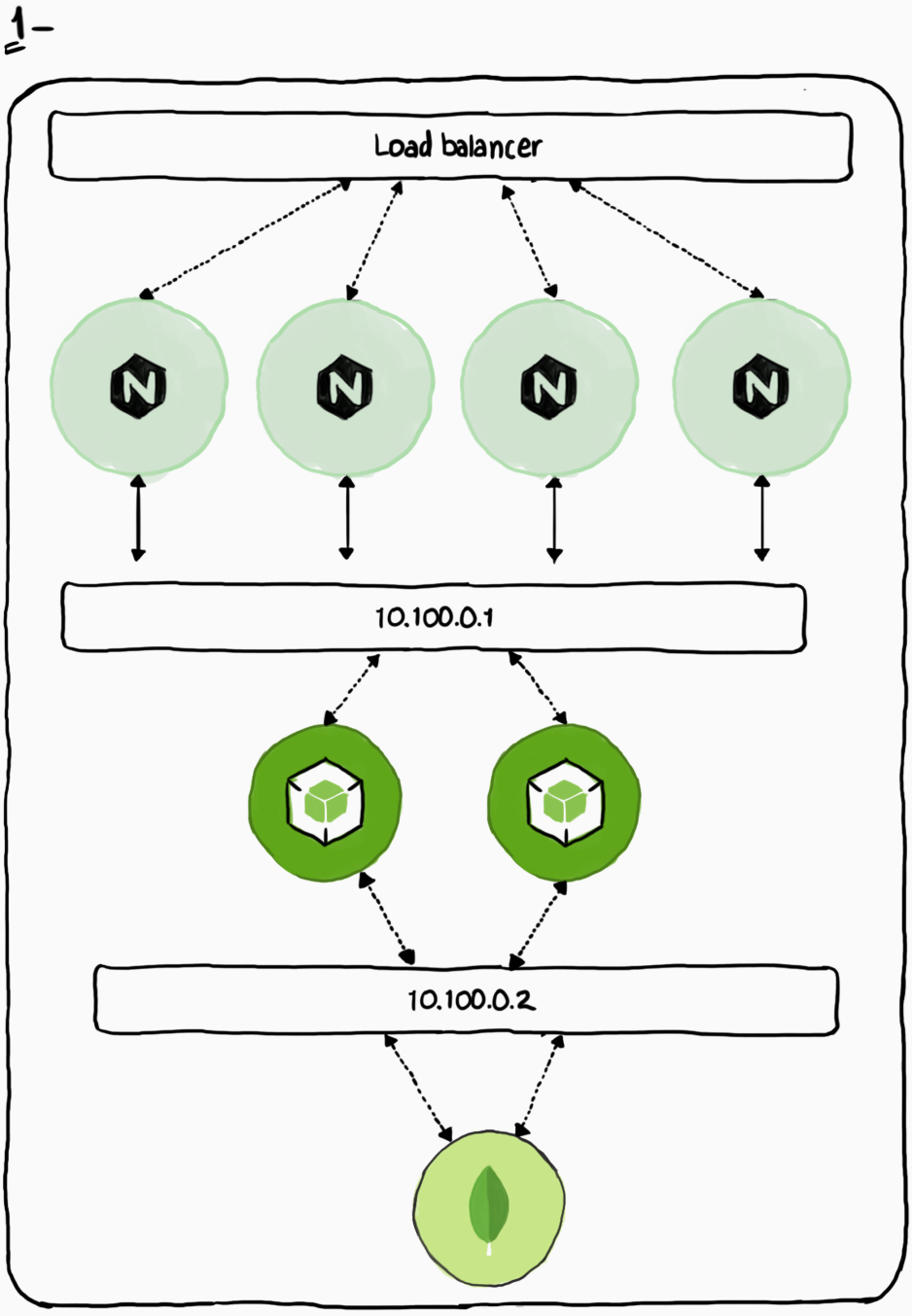
You don't need any Python knowledge here. Just notice that the GET request (line 3) is directed to 127.0.0.1. This is the loopback interface, but it is shared by the web server and the watcher containers. Communicating with localhost provides as low network latency as possible.
Since we have two containers in a pod, we will need to use the -c option with kubectl when we need to address a specific container. For example

```
kubectl exec -it webserver -c webwatcher -- /bin/bash
```

Summary

- Kubernetes pods are the foundation for all higher Kubernetes objects.
- A pod hosts one or more containers. It can be created using either a command or a YAML/JSON file.
- We use kubectl to create pods, view the running ones, modify their configuration, or terminate them. Kubernetes will attempt to restart a failing pod by default.
- If the pod fails to start indefinitely, we can simply use the kubectl describe command to discover what went wrong.
- Finally, we demonstrated the "how and why" regarding whether a pod could, or should, host more than one container.

3. Services, The Pods Interfaces



3.1 What is Kubernetes Service? Why Do We Need It?

As you know by this point, Pods are the basic building blocks of any Kubernetes cluster. They host one or more containers. A Kubernetes Service acts as a layer above the pods and it's always aware of the pods that it manages: their count, their internal IP addresses, the ports they expose and so on.

But how can pods reach one another? Consider the following example: you created a web application that contains four pods on the frontend, having Nginx as their application. The frontend pod needs to make a request to one of the backend NodeJS pods. If we have two, which one should the request be directed to? Additionally, pods get new IPs after each restart. An abstraction layer is needed to keep communication consistent across different pod instances. Kubernetes offers the Service object a solution to these situations. As seen in the illustration below, when Nginx receives an HTTP request, it is not aware of which of the NodeJS pods to direct the request to. A service will expose an interface that will route the request coming from Nginx to one of its pods, receive the response, and direct it back to the webserver.

3.2 Deploying A Kubernetes Service

Like all other Kubernetes objects, a service can be defined using a YAML or JSON file that contains the necessary definitions (they can also be created using just the command line, but this is not recommended). Let's create a NodeJS service definition - it may look like the following:

```
apiVersion: v1
kind: Service
metadata:
  name: backend
spec:
  selector:
    app: nodejs
  ports:
  - protocol: TCP
    port: 80
    targetPort: 3000
```

- The file starts with defining the API version it will use to contact the Kubernetes API server.

- Then, it defines the kind of object that it intends to manage: a service.
- The metadata contains the name of this service. Later on, applications will use this same name to communicate with the service.
- The spec part defines a selector - this is where we inform the service which pods will be under its control. Any pod that has a label "app=nodejs" will be handled by our service.
- The spec also defines how our service will handle the network in the ports array. Each port will have a protocol (TCP in our example, but services support UDP and other protocols), a port number that will be exposed, and a targetPort on which the service will contact the target pod(s). In our example, the pod will be available on port 80, but it will reach its pods on port 3000 (handled by NodeJS).

Applying this service definition (and all other service definitions) can be done using kubectl as follows:

```
kubectl apply -f definition.yaml
```

Here *definition.yaml* is the YAML file that contains the instructions.

3.3 How Can Kubernetes Services Expose More Than One Port?

Kubernetes Services allow you to define more than one port per service definition. Let's see how a web server service definition file may look:

```
apiVersion: v1
kind: Service
metadata:
  name: webserver
spec:
 selector:
   app: web
  ports:
  - name: http
    port: 80
    targetPort: 80
  - name: https
    port: 443
    targetPort: 443
```

Notice that if you are defining more than one port in a service, you must provide a name for each port so that they are recognizable.

3.4 Can We Use A Kubernetes Service Without Pods?

While the traditional use of a Kubernetes Service is to abstract one or more pods behind a layer, services can do more than that. Consider the following use cases where services do not work on pods:

- You need to access an API outside your cluster (examples: weather, stocks, currency rates).
- You have a service in another Kubernetes cluster that you need to contact.
- You need to shift some of your infrastructure components to Kubernetes. But, since you're still evaluating the technology, you need it to communicate with some backend applications that are still outside the cluster.
- You have another service in another namespace that you need to reach

The commonality here is that the service will not be pointing to pods - it'll be communicating with other resources inside or outside your cluster. Let's create a service definition that will route traffic to an external IP address:

```
apiVersion: v1
kind: Service
metadata:
  name: external-backend
spec:
  ports:
  - protocol: TCP
    port: 3000
    targetPort: 3000
```

Here, we have a service that connects to an external NodeJS backend on port 3000. But, this definition does not have pod selectors. It doesn't even have the external IP address of the backend! So, how will the service route traffic?

Normally, a service solves this problem by using an Endpoint object (behind the scenes) to map the IP addresses of the pods that match its selector.

3.5 What Is An 'Endpoint' Object In Terms Of Kubernetes?

Endpoints are used to track which pods are available so that the service can direct traffic to them. Yet, here we're not using pods at all. So, we'll need to manually create an endpoint object. Let's take a look at the following Endpoint definition file to see how it works:

```
apiVersion: v1
kind: Endpoints
metadata:
  name: external-backend
subsets:
  - addresses:
      - ip: 159.76.214.243
    ports:
      - port: 3000
```

You will need to apply this endpoint definition file then create the service. Now, any traffic arriving at our service on port 80 will be automatically routed to 159.76.214.243:3000. This may be your NodeJS backend that is running outside the cluster. As a side note, you cannot use any of the loopback IP addresses (127.0.0.0/8) or one of the link-locals (169.254.0.0/16 and 224.0.0.0/24) as the destination IP address. Additionally, the destination IP address cannot be the cluster IP of another Kubernetes Service. More on the service cluster IP later in this chapter.

3.6 But My Pods Already Have Internet Access, Why Use A Service

Consider that you have hundreds of pods, all of them are designed to contact that NodeJS server on 159.76.214.243:3000. Now, what if the server started using a different IP? There are many reasons why IP addresses get changed. You will have to manually modify all your containers' YAML files to point to the new IP address. Using a Kubernetes Service as a proxy will allow you to simply make this change once (and only in one place).

3.7 What If The Remote Application Uses A DNS Name?

Kubernetes Services can also connect to external servers by specifying DNS names rather than IP addresses. Those are referred to as ExternalName services. The following service definition will route traffic to an external API:

```
apiVersion: v1
kind: Service
metadata:
  name: weather
spec:
  type: ExternalName
  externalName: api.weather.com
  ports:
  - port: 80
```

Any web requests going to http://weather will automatically be routed to api.weather.com.
Note that the externalName can also be the DNS name of another service in another namespace. For example, `externalName: middleware.prod.svc.cluster.local`
Additionally, if you're deploying an ExternalName service in your cluster, you must use DNS as the service discovery method to be able to contact it. We'll break down service discovery methods in the next section.

3.8 What Is Service Discovery In Kubernetes?

Let's revisit our web application example. You're writing the configuration files for Nginx and you need to specify an IP address or URL for whichever web server will route backend requests. For demo purposes, here's a sample Nginx configuration snippet for proxying requests:

```
server {
  listen 80;

  server_name myapp.example.com;

  location /api {
      proxy_pass http://??/;
  }
}
```

The proxy_pass part here must point to the service's IP address or DNS name to be able to reach one of the NodeJS pods. In Kubernetes, there are two ways to discover services: environment variables or DNS. Next, we'll talk about each in a bit more detail.

3.9 Service Discovery Through Environment Variables

When a pod is scheduled to a node, the kubelet provides the pod with the necessary information to access services through environment variables. If we have a service backend that exposes port 3000 and was assigned an IP of 10.0.0.20, Kubernetes will automatically export environment variables as:

```
BACKEND_SERVICE_HOST=10.0.0.20
BACKEND_SERVICE_PORT=3000
```

This is not the most reliable way to discover services. In order to inject those environment variables in the pod, the service must be created before the pod. So, if the service was recreated for any reason after the pods are already running, they won't have access to those environment variables and service discovery will fail.

Also, not all applications support injecting environment variables in their configurations. For example, Nginx does not recognize environment variables in its configuration files out of the box. Fortunately, there are some workarounds for this, though.

3.10 Service Discovery Through DNS

It is highly recommended to use DNS for service discovery. A DNS component like CodeDNS will always listen for any newly created services by constantly contacting the API server of the cluster. Once it detects the presence of a new service, it creates the necessary records so that pods can communicate with it through a URL.

So, back to our example, if our service backend was created in the middleware namespace, it can be accessed by pods in the same namespace by talking to http://backend or http://backend:3000 (if it was not listening on port 80). If the client pod was created in another namespace, then it must use the fully qualified name for the service eg., http://backend.middleware.

So, now our Nginx configuration can have this line and work perfectly:

```
      proxy_pass http://backend.middleware/;
```

3.11 Kubernetes Services Connectivity Methods

If you've reached this far, you're now able to contact your services by name. Whether you're using environment variables or you've deployed a DNS, you get the service name resolved to an IP address. Now you want to get serious and make it accessible from outside your cluster? There are three ways to do that:

ClusterIP

The ClusterIP is the default service type - Kubernetes will assign an internal IP address to your service. This IP address is reachable only from inside the cluster. You can (optionally) set this IP in the service definition file. Think back to the earlier case where you had a DNS record that you didn't want to change, but you wanted the name to resolve to the same IP address. Now, you can easily do this by defining the clusterIP part of the service definition as follows:

```
apiVersion: v1
kind: Service
metadata:
  name: external-backend
spec:
  ports:
  - protocol: TCP
    port: 3000
    targetPort: 3000
  clusterIP: 10.96.0.1
```

However, you can't just add any IP address. It must be within the service-cluster-ip-range, which is a range of IP addresses assigned to the service by the Kubernetes API server. You can get this range through a simple kubectl command:

```
kubectl cluster-info dump  | grep service-cluster-ip-range
```

You can also set the clusterIP to none, effectively creating a Headless Service..

3.12 What Is The Use Of A Headless Service In Kubernetes?

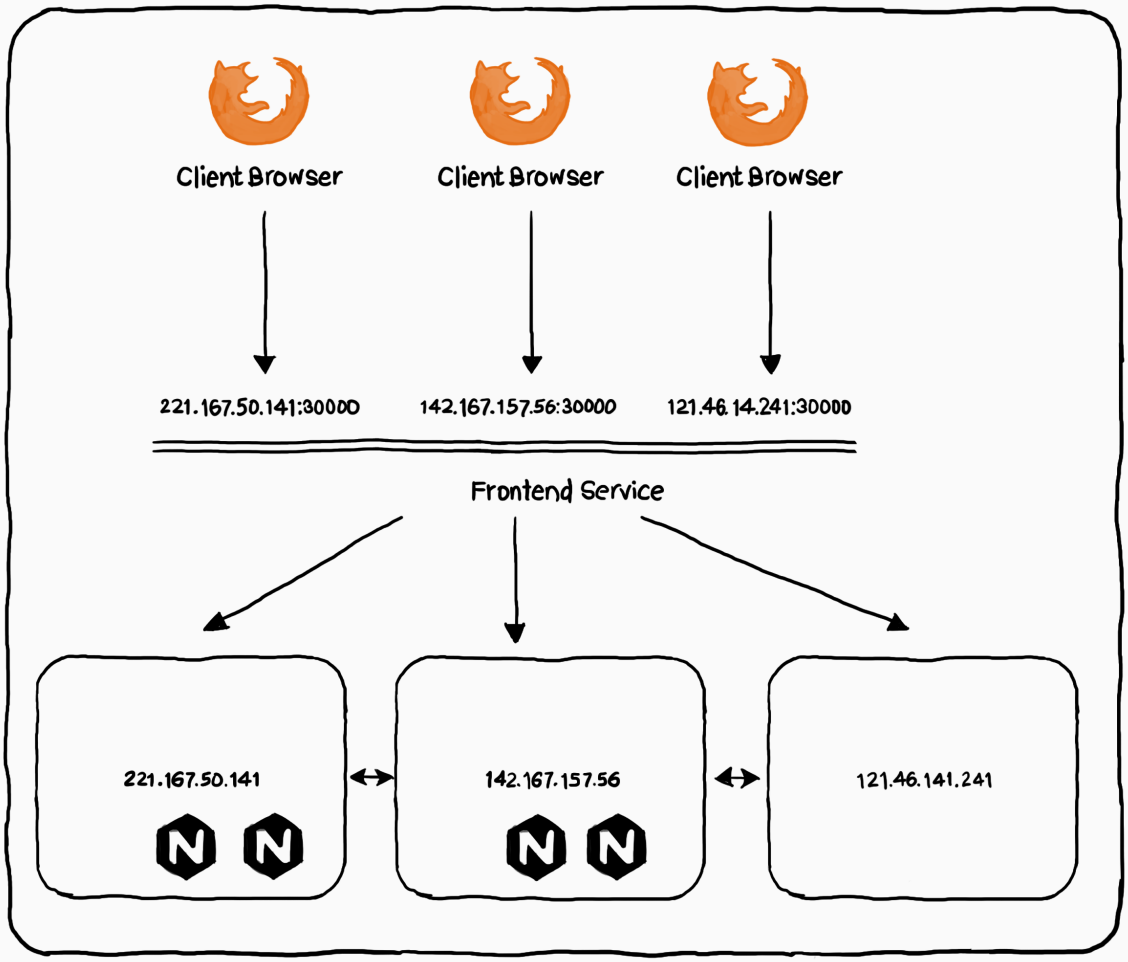
As mentioned, the default behavior of Kubernetes is to assign an internal IP address to the service. Through this IP address, the service will proxy and load-balance the requests to the pods behind. If we explicitly set this IP address (clusterIP) to none, this is like telling Kubernetes "I don't need load balancing or proxying, just connect me to the first available pod."

Let's look at a common use case. If you host, for example, your MongoDB on a single pod, you will need a service definition on top of it to take care of the pod being restarted and acquiring a new IP address. But you don't need any load balancing or routing. You only need the service to patch the request to the backend pod. Hence, the name, headless i.e. a service that does have an IP.

But, what if a headless service was created and is managing more than one pod? In this case, any query to the service's DNS name will return a list of all the pods managed by this service. The request will accept the first IP address returned. Obviously, this is not the best load-balancing algorithm - the bottom line, use a headless service when you need a single pod.

NodePort

This is one of the service types that are used when you want to enable external connectivity to your service. If you're going to have four Nginx pods, the NodePort service type is going to use the IP address of any node in the cluster combined with a specific port to route traffic to those pods. The following graph will demonstrate the idea:



You can use the IP address of any node, the service will receive the request and route it to one of the pods.

A service definition file for a service of type NodePort may look like this

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
spec:
  type: NodePort
  ports:
    - port: 80
      nodePort: 30000
      targetPort: 80
  selector:
    app: web
```

Manually allocating a port to the service is optional. If left undefined, Kubernetes will automatically assign one - they fall within the range of 30000-32767. If you're going to choose one yourself, ensure that the port is not already in use by another service. Otherwise, Kubernetes will report that the API transaction has failed.

Note that you must always anticipate the possibility of a node going down, and its IP address is no longer reachable. The best practice here is to place a load balancer above your nodes.

LoadBalancer

This service type works when you're using a cloud provider to host your Kubernetes cluster. When you choose LoadBalancer as the service type, the cluster will contact the cloud provider and create a load balancer. Traffic arriving at this load balancer will be forwarded to the backend pods. The specifics of this process is dependent on how each provider implements its load-balancing technology.

Different cloud providers handle load balancer provisioning differently. For example, some providers allow you to assign an IP address to the component, while others choose to assign short-lived addresses that constantly change. Kubernetes was designed to be highly portable. You can add loadBalancerIP to the service definition file. If the provider supports it, it will be implemented. Otherwise, it will be ignored. Let's take a look at a sample service definition that uses LoadBalancer:

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
spec:
  type: LoadBalancer
  loadBalancerIP: 78.11.24.19
  selector:
    app: web
 ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

One of the main differences between the LoadBalancer and the NodePort service types is, in the latter, you get to choose your own load balancing layer. You're not bound to the cloud provider's implementation.

So, let's take a look at the different connectivity methods that a Kubernetes service can use:

Name	IP address	Allowed access	Use cases
ClusterIP	The Service IP	Internal to the cluster	Sandboxed environments
NodePort	Any node IP + a designated port	External through one of the nodes	You need to implement your own load balancer and routing mechanism
LoadBalancer	The cloud provider's load balancer IP and port	External through the load balancer	You're in a cloud environment and don't need custom configurations

3.13 An Introduction To The Ingress Controller

Kubernetes Services offer three different ways for a network client to access them. However, you probably noticed that each service must define its connection type. For example, if we have three services (nodejs_service, flask_service, and spring_service) and each of them is hosting an API, then we need three different load balancers for each service. Therefore, we will have three IP addresses and three DNS records. To illustrate:

IP address	Resolves to
253.16.53.59	nodejs.example.com
233.119.165.52	flask.example.com
212.248.38.178	spring.example.com

While what we really need is:

253.16.53.59	www.example.com

And routing should be handled like:

URL	Gets routed to
www.example.com/api/v/1customers	nodejs_service
www.example.com/api/v/1invoices	flask_service
www.example.com/api/v/1employees	spring_serivce

To address this requirement, Kubernetes introduced the Ingress Controller. Note that Ingress is not part of the Service resource and is independent. The Ingress

Controller has its own definition - you can create it, modify it and destroy it regardless of the Service(s) it is handling.

Like the Load Balancer, an Ingress controller will receive external traffic and route it to each service based on the URL that is requested.

So, while you need a separate load balancer (or a NodePort) for each service you create, you only need one Ingress Controller to provide external access to your services.

Summary

- Kubernetes services provide the interface through which pods can communicate with each other. They also act as the main gateway for your application. Services use selectors to identify which pods they should control. They expose an IP address and a port that is not necessarily the same port at which the pod is listening. Services can expose more than one port, albeit they must have names in this case.
- Services can also route traffic to other services, external IP addresses, or DNS names. They can be discovered through environment variables, which get injected into the pods when they are started. Alternatively, a DNS can be deployed to the cluster and be used to track the services' names and IP addresses, which is the recommended service discovery method.
- Services use different ways to expose their IP addresses: the cluster IP, which is the default, where it is accessible inside the cluster network only, the NodePort, where the service uses the IP address of any node in the cluster, combined with a specific port, and the LoadBalancer, where the cloud provider is contacted (behind the scenes) to provision a load balancer that will route traffic to the pods.
- To address the drawbacks of having to create a load balancer or a NodePort resource for each service in the cluster, Kubernetes introduced the Ingress Controller. With an Ingress resource, you can route traffic to all of your services based on the URL requested - there will be one entry point for the application.

4. ReplicaSet

4.1 What Is A Kubernetes ReplicaSet?

A Kubernetes pod serves as a deployment unit for the cluster. It may contain one or more containers. However, containers (and accordingly, pods) are short-lived entities. A container hosting a PHP application, for example, may experience an unhandled code exception causing the process to fail, effectively crashing the container. Of course, the perfect solution for such a case is to refactor the code to properly handle exceptions. But, until that happens, we need to keep the application running and the business going. In other words, we need to restart the pod whenever it fails. In parallel, developers are monitoring, investigating and fixing any errors that make it crash. At some point, a new version of the pod is deployed, monitored and maintained. It's an ongoing process that is part of the DevOps practice.

Another requirement is to keep a predefined number of pods running. If more pods are up, the additional ones are terminated. Similarly, if one or more pods failed, new pods are activated until the desired count is reached.

A Kubernetes ReplicaSet resource was designed to address both of those requirements. It creates and maintains a specific number of similar pods (replicas). In this chapter, we'll discuss how we can define a ReplicaSet and what are the different options that can be used for fine-tuning it.

4.2 How Does ReplicaSet Manage Pods?

In order for a ReplicaSet to work, it needs to know which pods it will manage so that it can restart the failing ones or kill the unneeded. It also requires understanding how to create new pods from scratch in case it needs to spawn new ones.

A ReplicaSet uses labels to match the pods that it will manage. It also needs to check whether the target pod is already managed by another controller (like a Deployment or another ReplicaSet). So, for example, if we need our ReplicaSet to manage all pods with the label role=webserver, the controller will search for any pod with that label. It will also examine the ownerReferences field of the pod's metadata to

determine whether or not this pod is already owned by another controller. If it isn't, the ReplicaSet will start controlling it. Subsequently, the ownerReferences field of the target pods will be updated to reflect the new owner's data.

To be able to create new pods, the ReplicaSet definition includes a template part containing the definition for new pods. Next, we'll take a look at an example.

4.3 Your First ReplicaSet

Let's create a ReplicaSet that will ensure that we have four pods serving Nginx. A YAML definition file for this controller may look as follows:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: web
  labels:
     env: dev
     role: web
spec:
  replicas: 4
  selector:
     matchLabels:
       role: web
  template:
     metadata:
       labels:
         role: web
     spec:
       containers:
       - name: nginx
         image: nginx
```

Save the above in a file and give it a name, eg., nginx_replicaset.yaml. Next, let's see the effect of applying this definition before discussing what each line does:

```
kubectl apply -f nginx_replicaset.yaml
```

Now, if we run `kubectl get pods`, we should see an output similar to this:

```
NAME       READY   STATUS              RESTARTS   AGE
web-6n9cj    0/1      ContainerCreating   0      15s
web-7kqbm    0/1      ContainerCreating   0      15s
web-9src7    0/1      ContainerCreating   0      16s
web-fvxzf    0/1      ContainerCreating   0      15s
```

In a few moments, all of the pods should be in the running state.

If you have more than one ReplicaSet running, you may want to have an overview of their combined status. This can be done by issuing `kubectl get rs`. You should see an output like:

```
NAME   DESIRED   CURRENT   READY   AGE
web   4         4         4       2m
```

4.4 The ReplicaSet Definition File

Let's examine the definition file that was used to create our ReplicaSet:

- The apiVersion for this object is currently app/v1
- The kind of object is ReplicaSet
- In the metadata part, we name this ReplicaSet - we also define a number of labels to identify it.
- The spec part is mandatory in the ReplicaSet object. It defines:
  - The number of replicas this controller should maintain - it defaults to 1 (if otherwise not specified).
  - The selection criteria by which the ReplicaSet will choose its pods. Be careful not to use a label that is already in use by another controller. Otherwise, another ReplicaSet may acquire the pod(s) first. Also, notice that the labels defined in the pod template (spec.template.metadata.label) cannot be different than those defined in the matchLabels part (spec.selector).
  - The pod template is used to create (or recreate) new pods - it has its own metadata, and spec where containers are specified.

4.5 Is Our ReplicaSet The Owner Of Those Pods?

Okay, so we have four pods running, and our ReplicaSet reports that it is controlling four pods. In a busier environment, you may want to verify that a particular pod is actually managed by this ReplicaSet and not by another controller. By simply

querying the pod, you can get that info:

```
kubectl get pods web-6n9cj -o yaml | grep -A 5 owner
```

The first part of the command will get all the pod information, which may be too verbose. Using grep with the -A flag (it takes a number and prints that number of lines after the match) will get us the required information as in this example:

```
ownerReferences:
  - apiVersion: extensions/v1beta1
      blockOwnerDeletion: true
      controller: true
      kind: ReplicaSet
      name: web
```

4.6 How Can I Remove A Pod From A ReplicaSet?

You can remove (not delete) a pod that is managed by a ReplicaSet by simply changing its label. Let's isolate one of the pods created in our previous example:

```
kubectl edit pods web-44cjb
```

Once the YAML file is opened, change the pod label to be role=isolated or anything different than role=web. In a few moments, run `kubectl get pods`. You'll notice that we have five pods now. That's because the ReplicaSet dutifully created a new pod to reach the desired number of four pods. The isolated one is still running, but it is no longer managed by the ReplicaSet.

4.7 Scaling And Autoscaling ReplicaSets

You can easily change the number of pods a particular ReplicaSet manages in one of two ways:

1. 1. Edit the controller's configuration by using `kubectl edit rs ReplicaSet_name` and change the replicas count up or down as you desire.
2. Use kubectl directly - for example, `kubectl scale --replicas=2 rs/web`. Here, we're scaling down the ReplicaSet used in the chapter's example to manage two pods instead of four. The ReplicaSet will get rid of two pods to maintain the desired

count. If you followed the previous section, you may find that the number of running pods is three instead of two; as we isolated one of the pods so it is no longer managed by our ReplicaSet.

ReplicaSets can also be used to adapt the number of pods according to the node's CPU load. To enable autoscaling for our web ReplicaSet, we can use the following command:

```
kubectl autoscale rs web --max=5
```

This will use the Horizontal Pod Autoscaler (HPA) with the ReplicaSet to increase the number of pods when the CPU load gets higher, but it should not exceed five pods. When the load decreases, it cannot have less than the number of pods previously specified (two in our example).

4.8 The (In)Correct Way For A ReplicaSet To Adopt A Pod

The recommended practice is to use the ReplicaSet's template for creating and managing pods. However, because of the way ReplicaSets work, if you create a bare pod (not owned by any controller) with a label that matches the ReplicaSet selector, the controller will automatically adopt it. This has a number of undesirable consequences. Let's have a quick lab to demonstrate them.

Deploy a pod by using a definition file like the following:

```
apiVersion: v1
kind: Pod
metadata:
  name: orphan
  labels:
     role: web
spec:
  containers:
  - name: orphan
     image: httpd
```

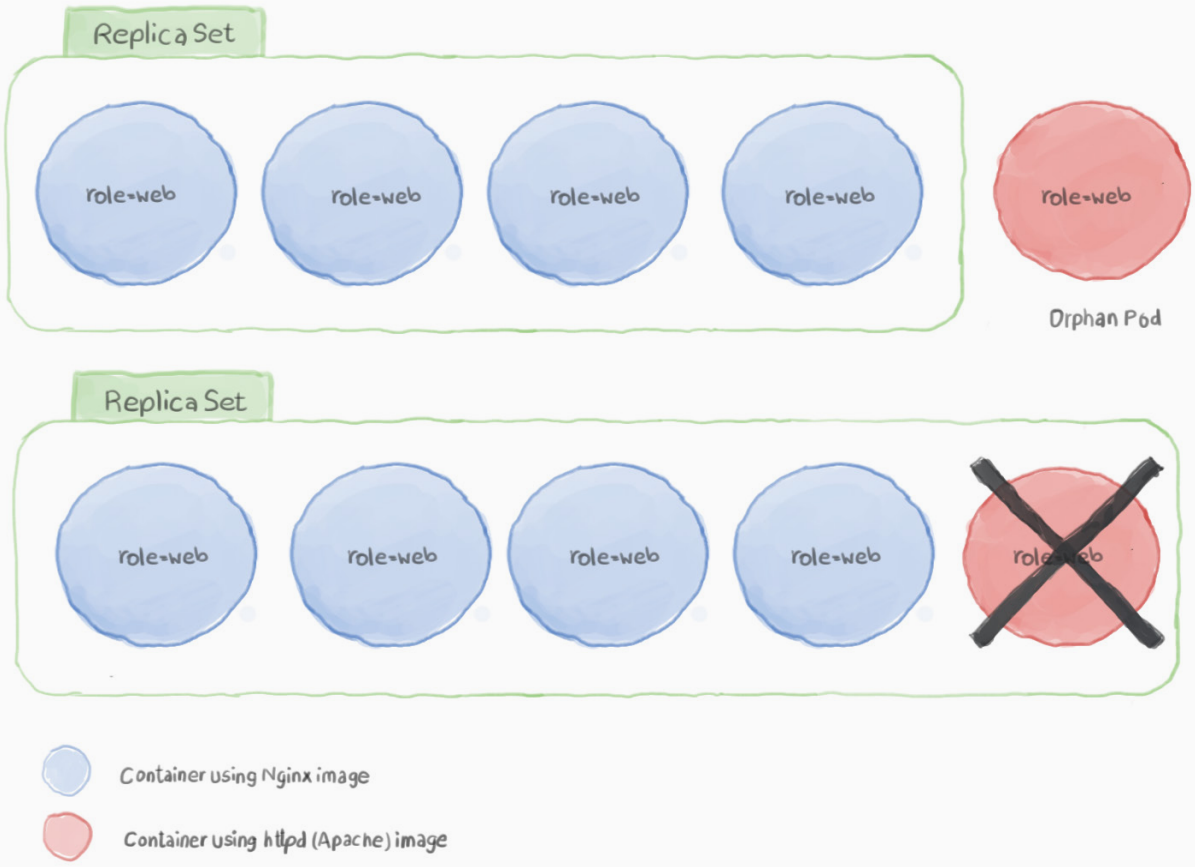
It looks a lot like the other pods, but it is using Apache (httpd) instead of Nginx for an image. Using kubectl, we can apply this definition as:

```
kubectl apply -f orphan.yaml
```

Give it a few moments for the image to get pulled, and the container is spawned, then run `kubectl get pods`. You should see the output looks like the following:

```
NAME        READY   STATUS        RESTARTS   AGE
orphan      0/1   Terminating   0          1m
web-6n9cj    1/1  Running     0          25m
web-7kqbm    1/1  Running     0          25m
web-9src7    1/1  Running     0          25m
web-fvxzf    1/1  Running     0          25m
```

The pod is being terminated by the ReplicaSet because, by adopting it, the controller has more pods than it was configured to handle. So, it is killing the excess one.



Another scenario where the ReplicaSet won't terminate the bare pod is that the latter gets created before the ReplicaSet does. To demonstrate this case, let's destroy our ReplicaSet:

```
kubectl delete -f nginx_replicaset.yaml
```

Now, let's create it again (while our orphan pod is still running):

```
kubectl apply -f nginx_replicaset.yaml
```

Let's have a look at our pods status by running `kubectl get pods`. The output should look like this:

```
NAME       READY   STATUS   RESTARTS   AGE
orphan     1/1   Running   0          29s
web-44cjb   1/1  Running   0          12s
web-hcr9j   1/1  Running   0          12s
web-kc4r9   1/1  Running   0          12s
```

Now, the situation is that we have three pods running Nginx, and one pod running Apache (the httpd image). As far as the ReplicaSet is concerned, it is handling four pods (the desired number), and their labels match its selector. But what if the Apache pod went down? Let's do just that:

```
kubectl delete pods orphan
```

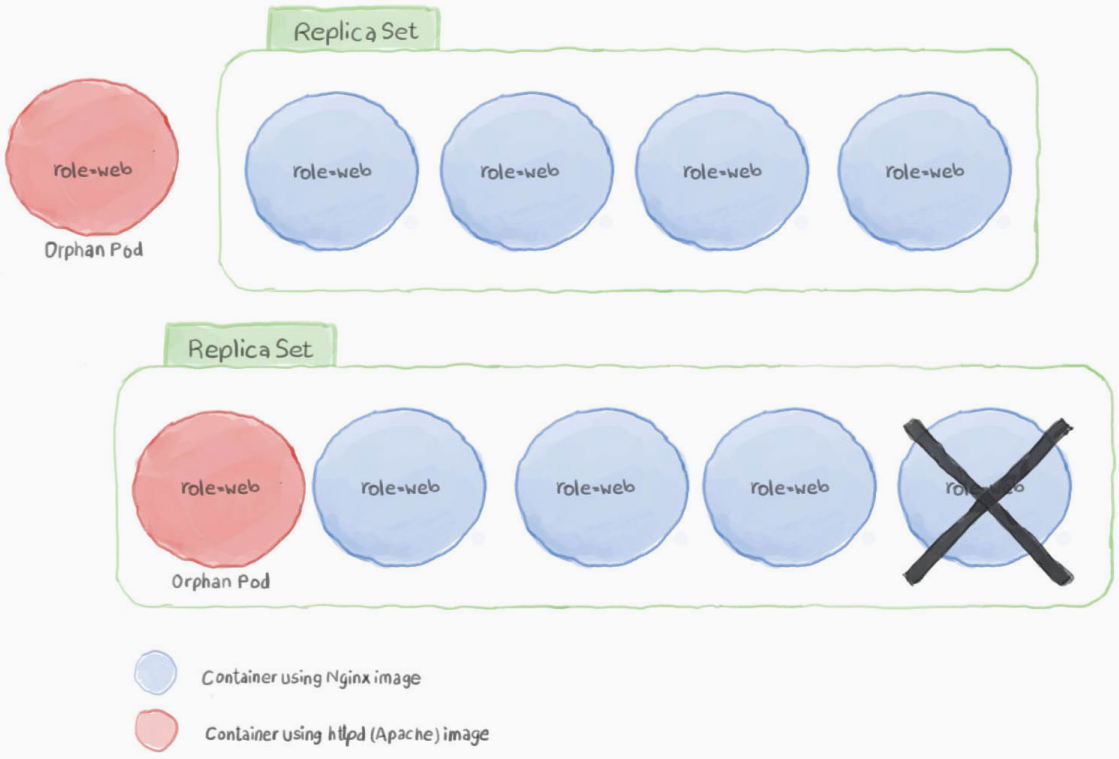
Next, let's see how the ReplicaSet responded to this event:

```
kubectl get pods
```

The output should be something like:

```
NAME       READY   STATUS              RESTARTS   AGE
web-44cjb   1/1  Running          0          24s
web-5kjwx   0/1  ContainerCreating   0       3s
web-hcr9j   1/1  Running          0          24s
web-kc4r9   1/1  Running          0          24s
```

The ReplicaSet is doing what it was programmed to: creating a new pod to reach the desired state using the template that was added in its definition. Obviously, it is creating a new Nginx container instead of the Apache one that was deleted.



So, although the ReplicaSet is supposed to maintain the state of the pods it manages, it failed to respawn the Apache webserver. It replaced it with an Nginx one.

The bottom line: you should never create a pod with a label that matches the selector of a controller unless its template matches the pod definition. The recommended procedure is to always use a controller like a ReplicaSet or, even better, a Deployment to create and maintain your pods.

4.9 Deleting ReplicaSets

As with other Kubernetes objects, a ReplicaSet can be deleted by issuing a kubectl command like the following:

```
kubectl delete rs ReplicaSet_name
```

Alternatively, you can also use the file that was used to create the resource (and possibly, other resource definitions as well) to delete all the resources defined in the file as follows:

```
kubectl delete -f definition_file.yaml
```

The above commands will delete the ReplicaSet and all the pods that it manages. But sometimes you may want to just delete the ReplicaSet resource, keeping the pods unowned (orphaned). Maybe you want to manually delete the pods and you don't want the ReplicaSet to restart them - this can be done using the following command:

```
kubectl delete rs ReplicaSet_name --cascade=false
```

If you run `kubectl get rs` now you should see that there are no ReplicaSets there. Yet if you run `kubectl get pods`, you should see all the pods that were managed by the destroyed ReplicaSet still running.

The only way to get those pods managed by a ReplicaSet again is to create this ReplicaSet with the same selector and pod template as the previous one. If you need a different pod template, you should consider using a Deployment instead, which will handle replacing pods in a controlled way.

Summary

- ReplicaSets are Kubernetes controllers that are used to maintain the number and running state of pods. It uses labels to select pods that it should be managing. A pod must be labeled with a matching label to the ReplicaSet selector, and it must not be owned by another controller so that the ReplicaSet can acquire it.
- Pods can be isolated from a ReplicaSet by simply changing their labels so that they no longer match the ReplicaSet's selector.
- ReplicaSets can be deleted with, or without, deleting their dependent pods.
- You can easily control the number of replicas (pods) the ReplicaSet should maintain through the command line, or by directly editing the ReplicaSet configuration on the fly.
- You can also configure the ReplicaSet to autoscale based on the amount of CPU load the node is experiencing.
- You may have read about ReplicationControllers in older Kubernetes documentation, articles or books. ReplicaSets are the successors of ReplicationControllers. They are recommended instead of ReplicationControllers as they provide more features

5. Deployments

5.1 Why Use A Kubernetes Deployment?

In the previous chapter, we discussed Kubernetes ReplicaSets. They're one of the controllers that manage one or more pods for you. However, ReplicaSets have a major drawback: once you select the pods that are managed by a ReplicaSet, you can't change their pod templates. So if you're using a ReplicaSet to deploy four pods with NodeJS running and you want to change the NodeJS image to a newer version, you need to delete the ReplicaSet and recreate it. Restarting the pods also causes downtime until the images are available and the pods are running again.

A Deployment resource uses a ReplicaSet to manage the pods - however, it handles updating them in a controlled way. Next, we'll dig deeper into Deployment Controllers and patterns.

5.2 Your First Deployment

Now, let's check out a quick demo of what Kubernetes Deployments can do. The following Deployment definition deploys four pods with Apache as their application host:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: apache-deployment
  labels:
    role: webserver
spec:
  replicas: 4
  selector:
    matchLabels:
      role: webserver
  template:
    metadata:
      labels:
        role: webserver
    spec:
      containers:
      - name: frontend
        image: httpd
        ports:
        - containerPort: 80
```

Now, save the above in a file. In this example, we named the file apache_deployment.yaml. Apply the definition to the cluster by running the following command:

```
kubectl apply -f apache_deployment.yaml --record
```

Note the use of the --record flag at the end of the command - while not required, this is a good practice that you should follow. The --record flag saves the command that issued the deployment in a list. Later on in the chapter, you'll see the value of keeping this information saved. In a few seconds, you can check the status of the pods by running `kubectl get pods`. Hint: you should see three pods running.

5.3 The Deployment Definition

Now, let's have a look at the definition file that we used to bring those pods up:

- The file starts with the apiVersion that accepts the Deployment API object. Currently, it's apps/v1.
- Then we have the type of resource: deployment.
- In the metadata, we define the name of this Deployment and its label.
- The spec field defines how many pods we need this Deployment to maintain. It also contains the selection criteria that the controller uses to acquire the target pods. The matchLabels field targets pods labeled role=webserver.
- The spec field also has the pod template that is used to create (or recreate) the pods.
- The spec.template.metadata defines the label that new pods will have.
- The spec.template.spec part has the actual container definition (owned by the container field). In our example, we define the container name and the port it listens on (HTTP 80).

5.4 Performing Updates with Zero Downtime (Deployment Rolling Updates)

So far, everything our Deployment has done is no different than a typical ReplicaSet.

The real power of a Deployment lies in its ability to update the pod templates without causing application outage.

Let's say that you have finished testing the Apache server version 2.4, and you are ready to use it in production. The current pods are using the older Apache version 2.0. The following command changes the deployment pod template to use the new image:

```
kubectl set image deployment apache-deployment apache=httpd:2.4
```

The above command changes the image tag of the containers named apache to use the image httpd tagged 2.4 instead of 2. An alternative way to achieve this is by editing the deployment configuration YAML directly using a command like the following:

```
kubectl edit deployment apache-deployment
```

Then, scroll down to the pod template part and change the httpd image tag. Once you've saved the configuration, the Deployment starts updating the pods one by one. You can see the actual progress of this operation by issuing the following command:

```
kubectl rollout status deployment apache-deployment
```

The output shows the update progress until all the pods use the new container image.

The algorithm that Kubernetes Deployments uses when updating will keep at least 25% of the pods running. Accordingly, it doesn't kill old pods unless a sufficient number of new ones are up. In the same sense, it does not create new pods until enough pods are no longer running. Through this algorithm, the application is always available during updates.

You can use the following command to determine the update strategy that the Deployment is using:

```
kubectl describe deployments | grep Strategy
```

The output should look like:

```
StrategyType:           RollingUpdate
RollingUpdateStrategy:  25% max unavailable, 25% max surge
```

We used grep here to filter out the command's output to reveal how it updates

the pods. If we remove the filter, we'll find some valuable information about the deployment's steps. Let's see:

```
Events:
  Type    Reason             Age    From                   Message
  ----    ------             ----   ----                   -------
  Normal  ScalingReplicaSet  28m    deployment-controller  Scaled up
replica set apache-deployment-6bdd4b58db to 4
  Normal  ScalingReplicaSet  4m38s  deployment-controller  Scaled up
replica set apache-deployment-67fd555f74 to 1
  Normal  ScalingReplicaSet  4m38s  deployment-controller  Scaled down
replica set apache-deployment-6bdd4b58db to 3
  Normal  ScalingReplicaSet  4m38s  deployment-controller  Scaled up
replica set apache-deployment-67fd555f74 to 2
  Normal  ScalingReplicaSet  4m34s  deployment-controller  Scaled down
replica set apache-deployment-6bdd4b58db to 2
  Normal  ScalingReplicaSet  4m34s  deployment-controller  Scaled up
replica set apache-deployment-67fd555f74 to 3
  Normal  ScalingReplicaSet  4m33s  deployment-controller  Scaled down
replica set apache-deployment-6bdd4b58db to 1
  Normal  ScalingReplicaSet  4m33s  deployment-controller  Scaled up
replica set apache-deployment-67fd555f74 to 4
  Normal  ScalingReplicaSet  4m32s  deployment-controller  Scaled down
replica set apache-deployment-6bdd4b58db to 0
```

You should find the following info at the end of the command output:

- it shows how the Deployment first created the ReplicaSet with four pods.
- then it used a new ReplicaSet with just one pod.
- immediately after, it kills one of the pods of the old ReplicaSet.

As you can see, it keeps killing pods from the old ReplicaSet and scaling up the new one until it replaces all pods.

Let's double-check that we have two ReplicaSets created for us by running kubectl get rs. The output should be similar to the following::

```
NAME                           DESIRED   CURRENT   READY   AGE
apache-deployment-67fd555f74   4         4         4       19m
apache-deployment-6bdd4b58db   0         0         0       43m
```

The old ReplicaSet has no pods, while the new one has all the four pods.

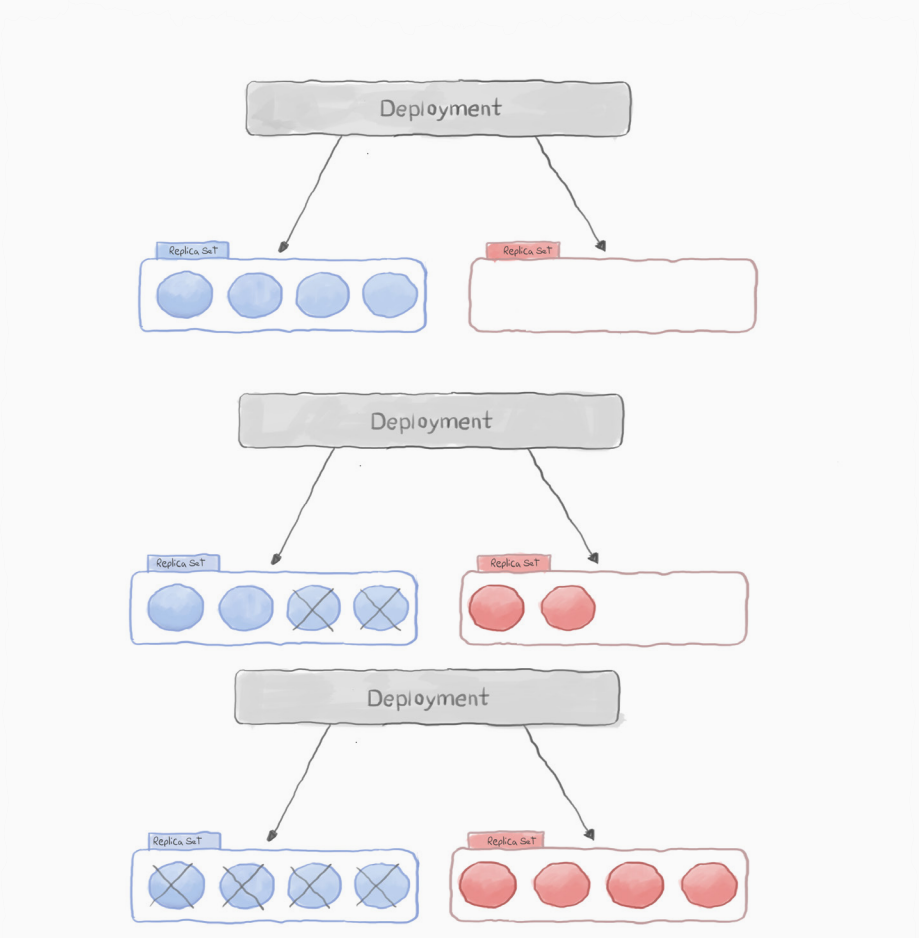
5.5 Kubernetes Deployments Strategies Overview

Rolling Update

As we mentioned previously, the rolling update is the default strategy that Kubernetes Deployments use. If you want to use the rolling update strategy, you needn't specify any parameters in the definition file. However, you may want to fine-tune how Kubernetes handles the transition of old pods to new ones. For example, Kubernetes automatically decides that it needs to keep at least 75% of the pods available. If you want to override this behavior, you can add .spec.strategy.type as follows:

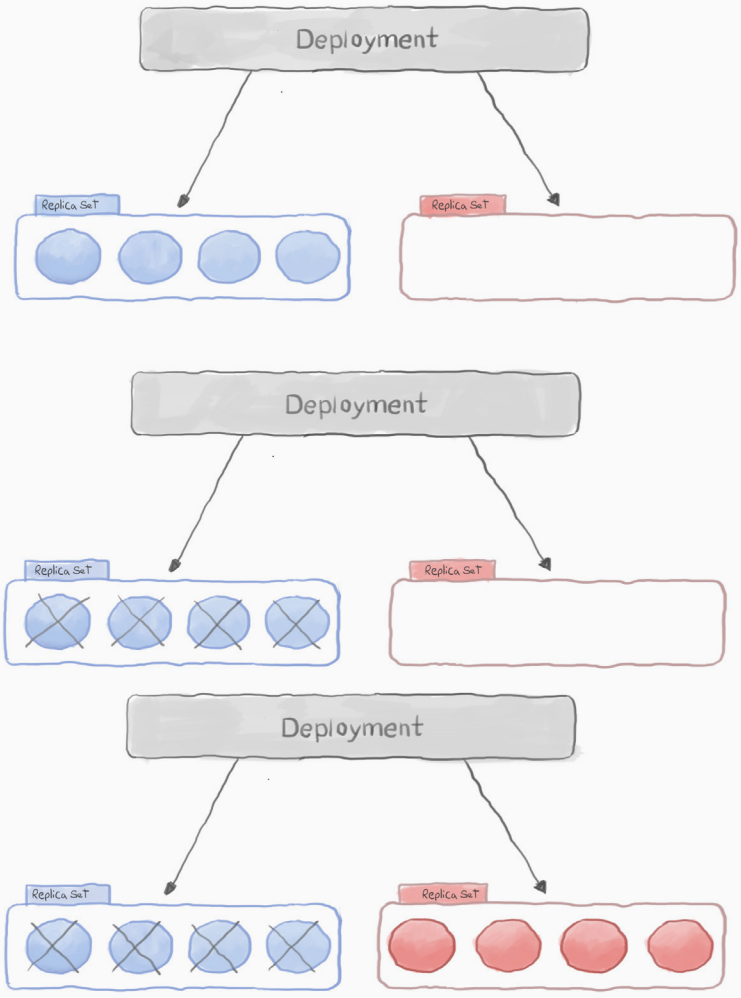
```
spec:
 strategy:
 type: RollingUpdate
 rollingUpdate:
   maxSurge: 1
   maxUnavailable: 50%
```

By setting the maxUnavailable parameter to 50%, we want Kubernetes to bring down as much as half of the running Pods during the update (this number can either be a percentage or a whole number).



Recreate Update

The Recreate strategy will bring down all of the old pods immediately and then start a new one (potentially causing unnecessary downtime). For example, if you discover a serious security flaw in your application and you need to immediately switch to the new patched image. You don't need any of your clients to use the old vulnerable version as this may negatively affect your business reputation. A Recreate deployment strategy is required here even if it brings the application down for a few moments. Perhaps you can display a friendly "under maintenance" message till the update is done.



There are other types of deployment strategies that engineers may need to use. At the time of this writing, the Kubernetes Deployment only supports the rolling update and recreate strategies. However, with the help of other controllers like services you can achieve more complex scenarios like:

- Blue/Green deployment: where you have two different versions of your application, the latest (green) and the running (blue). Once the green deployment is ready, you can configure the Service to select the new pods (through Pod Selector). If everything goes without issue, you can also update the blue version to the latest and use it as a staging environment.
- Canary release: named after the safety technique coal miners used to follow in the old days. They brought a cage containing some Canary birds and placed it at the entrance of the mine. If the birds died, that indicated the presence of toxic Carbon Monoxide emissions. Otherwise, the miners were good to go. In software release, the Canary type entails directing a subset of your users to the new version of your application and testing their feedback. The majority of the users are still using the old stable version. If no issues are detected, more and more users get directed to the new version until it's fully released. In Kubernetes, this can be done by creating another deployment with a smaller replica count (the Canary instance). The Deployment can be scaled up, or totally terminated, according to the test results.

5.6 Updating A Deployment While Another Is In Progress (Rollover Updates)

As soon as the Deployment controller detects a change in the pod template (i.e., an update), it creates a new ReplicaSet and starts rolling out the pods to that new ReplicaSet until all have been moved. However, sometimes, you may want to issue a new update while the existing one is still in progress. Let's take a look at an example.

Suppose you are updating ten application pods to version 1.1 (image myapp:1.1). And, while the update is in progress, the QA team informed you that they'd just finished testing version 1.2 and it's ready for deployment. So, you decide to interrupt the running update and go ahead with the latest version (image myapp:1.2). Behind the scenes, the Deployment was using a new ReplicaSet and had already moved three pods to use the myapp:1.1 image when it detected a new deployment request. It immediately creates another ReplicaSet, kills the three pods that moved, and starts

scaling up the newest ReplicaSet with pods using myapp:1.2. In other words, it does not wait for the whole ten pods to finish upgrading to myapp:1.1 before starting to migrate them to myapp:1.2. Instead, it aborts the existing operation and starts the new one right away. Such an operation is called rollover update, and it's a powerful technique to ensure that your pods are always in the desired state in the shortest time possible, and with no downtime.

5.7 Undoing A Deployment (AKA Rolling Back)

Kubernetes deployments allow you to roll back updates. There are many scenarios when you might want to undo a change. Let's say that customers started complaining about a bug that was not detected during the QA phase and you need to get the application back to the previous version until the bug gets fixed. For example, let's say that we decided to use Nginx instead of Apache for your web tier. You issued the following command to make that change:

```
kubectl set image deployment apache-deployment frontend=nginx:1.7.9
--record
```

After a few moments, all the pods were using Nginx as their web servers. Then, you realized that there are performance issues with the application, and clients are starting to complain. You need to configure the pods to use Apache again and there should be no downtime in this rollback process.

The Kubernetes Deployment controller keeps track of every Deployment that has been made (up to a configurable limit). Kubernetes considers changes in the pod template, and keeps only those in history. If the change was scaling up or down the number of running pods, it would not count as a record.

Back to our example. To rollback the Deployment to a previous one, you need first to list the last changes. The following command outputs the deployment history:

```
kubectl rollout history deployment apache-deployment
```

You should see the following output:

```
REVISION  CHANGE-CAUSE
1         kubectl apply --filename=apache_deployment.yaml --record=true
2         kubectl set image deployment apache-deployment
frontend=nginx:1.7.9 --record=true
```

The change-cause contains the command that was issued and caused the change. If you didn't use the --record flag, this field would equal None.

To roll back the latest Deployment and return to the previous state, run the following command:

```
kubectl rollout undo deployment apache-deployment
```

Kubernetes Deployment starts a process similar to what it used when upgrading the pods to Nginx. In a few moments, all the pods are running Apache again.

Occasionally you may want to rollback to a specific deployment - let's say that you made an upgrade from httpd image 2.4 to 2.4.39 before changing to Nginx and you want to revert to using httpd:2.4. That's two deployments back. You can specify the exact revision number you want your Deployment to roll back to by using the --to-revision flag. For example:

```
kubectl rollout undo deployment apache-deployment --to-revision=1
```

5.8 Scaling And Autoscaling Deployments

Since Deployments use ReplicaSets internally to manage pods, they also support scaling up or down. Let's scale our apache-deployment to run six pods instead of four:

```
kubectl scale deployment apache-deployment --replicas=6
```

If you check the pods now using kubectl get pods, you'll see the Deployment is creating two more pods.

You can also use Horizontal Pod Autoscaling (HPA) to automatically increase or decrease the number of pods in a deployment based on the CPU load on the node with the following command:

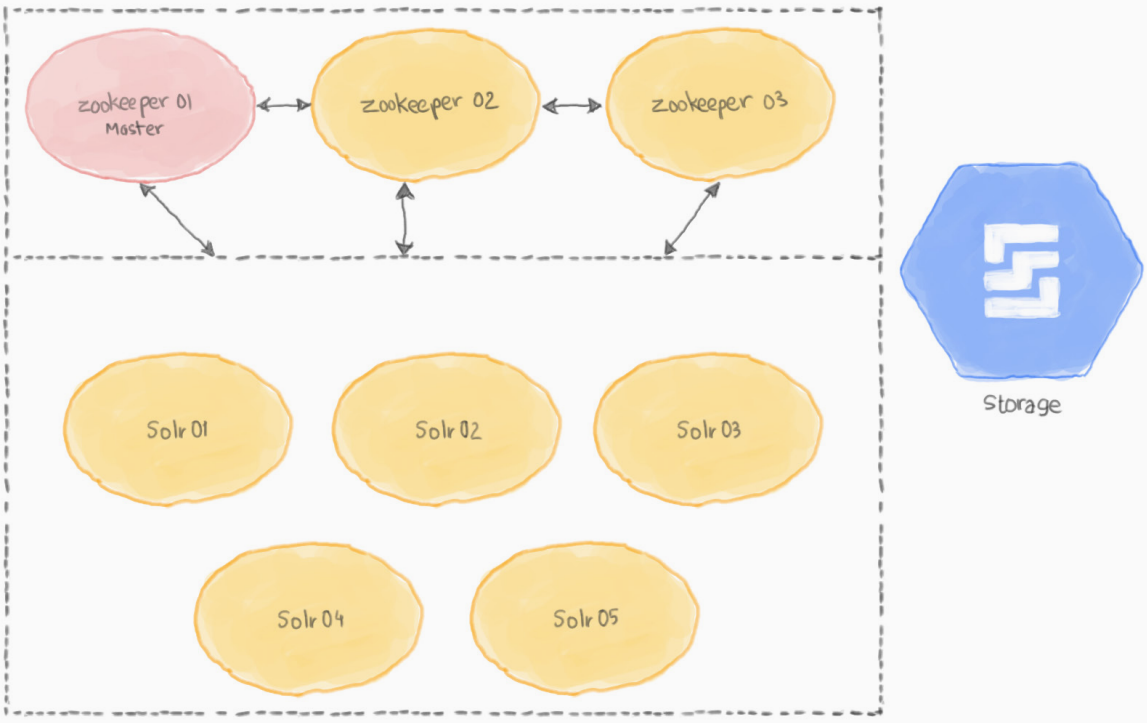
```
kubectl autoscale deployment apache-deployment --min=6 --max=10 --cpu-
percent=70
```

As you can see, it adds or kills pods from the Deployment according to the amount of CPU load on the node. It ensures that pods have an average CPU load of 70%. As the load increases, the Deployment spawns more pods until it reaches a max of ten. When the load is less, the Deployment kills extra pods as long as their number is not less than six. You can read more about the algorithm the Deployment uses for autoscaling here.

Summary

- Kubernetes Deployments are one of the most potent controllers you can use. They not only maintain a specified number of pods, but also ensure that any updates you want to make do not cause downtime.
- Behind the scenes, Deployments use ReplicaSets to manage pods.
- Kubernetes Deployments support rollover updates in which you can interrupt an ongoing deployment update and instruct the Deployment controller to start the new update immediately without causing any application outage.
- Kubernetes maintains a list of the recent deployments. You can use this list to rollback an update and you can also choose a specific deployment to roll back to by specifying its revision number.
- You can use Deployments to scale up, or down, the number of pods it is managing. You can also configure it to respond to CPU load by creating, or killing pods, subject to a maximum and minimum number.

6. StatefulSets, State Of The Pods



6.1 The Difference Between A Statefulset And A Deployment

A Statefulset is a Kubernetes controller that is used to manage and maintain one or more Pods. However, other controllers like ReplicaSets and Deployments do handle the same functions. So what exactly does Kubernetes use StatefulSets for? To answer this question, we need to discuss stateless versus stateful applications.

A stateless application is one that doesn't care which network it's using, and it doesn't need permanent storage - examples of stateless apps include web servers

like Apache, Nginx, or Tomcat.

On the other hand, we have stateful apps. Let's say you have a Solr database cluster that is managed by several Zookeeper instances. For such an application to function correctly, each Solr instance must be aware of the Zookeeper instances that are controlling it. Similarly, the Zookeeper instances themselves establish connections between each other to elect a master node. Due to such a design, Solr clusters are an example of stateful applications. If you deploy Zookeeper on Kubernetes, you'll need to ensure that pods can reach each other through a unique identity that does not change (hostnames, IPs, etc.). Other examples of stateful applications include MySQL clusters, Redis, Kafka, MongoDB, etc.

Given this difference, a Deployment is more suited to work with stateless applications. As far as a Deployment is concerned, Pods are interchangeable. Meanwhile, a StatefulSet keeps a unique identity for each Pod it manages and it uses the same identity whenever it needs to reschedule those Pods.

6.2 Exposing A Statefulset

A Kubernetes Service acts as an abstraction layer to pods. In a stateless application like an Nginx web server, the client does not (and should not) care which pod receives a response to the request. The connection reaches the Service, and it routes it to any backend pod. This is not the case in stateful apps. In the above diagram, a Solr pod may need to reach the Zookeeper master, not just any pod. For this reason, part of the Statefulset definition entails a Headless Service. Headless Service does not contain a ClusterIP - instead, it creates several Endpoints that are used to produce DNS records and each DNS record is bound to a pod. All of this is done internally by Kubernetes, but it's good to have an understanding of how it all functions.

6.3 How To Deploy A Stateful Application Using Kubernetes Statefulset?

In the below code snippet we are deploying a stateful application. For simplicity, we are using Apache as the pod image. The deployment is made up of three Apache web servers; all of them are connected to a persistent volume. Now, we'll create a new file and name it apache_stateful.yaml. Add the following to it:

```
---
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: www-disk
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
  zone: us-central1-a
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: webapp
spec:
  selector:
      matchLabels:
      app: web
  serviceName: web-svc
  replicas: 3
  template:
      metadata:
      labels:
      app: web
      spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: web
      image: httpd:2.4
      ports:
      - containerPort: 80
            name: http
      volumeMounts:
      - name: www
            mountPath: /var/www/html
  volumeClaimTemplates:
  - metadata:
      name: www
      annotations:
      volume.beta.kubernetes.io/storage-class: www-disk
      spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
      requests:
            storage: 1Gi
```

Before we start discussing the details of this definition, notice that the file actually contains two definitions: the storage class that the StatefulSet is using, and the StatefulSet itself.

Storage Class

Storage classes are Kubernetes objects that let the users specify which type of storage they need from the cloud provider. Different storage classes represent various service quality (eg., disk latency and throughput), and are selected depending on the scenario they are used for and the cloud provider's support. Persistent Volumes and Persistent Volume Claims use Storage Classes.

Persistent Volumes And Persistent Volume Claims

Persistent volumes act as an abstraction layer to save the user from going into the details of how storage is managed and provisioned by each cloud provider (in this example, we are using Google GCE). By definition, StatefulSets are the most frequent users of Persistent Volumes since they need permanent storage for their pods.

A Persistent Volume Claim is a request to use a Persistent Volume. If we use the Pods and Nodes analogy, then consider Persistent Volumes as the "nodes" and Persistent Volume Claims as the "pods" that use the node resources. The resources we're talking about here are storage properties, such as storage size, latency, throughput, etc.

Provisioning The StatefulSet

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: webapp
```

The definition starts with the API version that recognizes StatefulSets, the kind of the resource, and the metadata. In this example, we only added the resource name to the metadata, but you can also add labels to identify it further.

```
spec:
  selector:
     matchLabels:
     app: web
  serviceName: web-svc
```

The next thing we have is the spec, and the spec contains the pod selector. Like other Kubernetes controllers (eg., Deployments and ReplicaSets), StatefulSets use matchLabels to choose and acquire the pods that it should manage.

We also have a reference to a service that we are yet to create, the Headless Service that provides access to our pods.

```
  replicas: 3
```

The replicas object defines how many pods this StatefulSet will create.

The pod template defines how the pods get created, and it contains the pods' labels (they must match the pod selector).

```
  template:
      metadata:
      labels:
      app: web
      spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: web
      image: httpd:2.4
      ports:
      - containerPort: 80
            name: http
      volumeMounts:
      - name: www
            mountPath: /var/www/html
```

The pod template spec part contains the terminationGracePeriodSeconds. This setting defines how long Kubernetes should wait after sending the shutdown signal to the pod before force-deleting it. Kubernetes strongly recommends that you do not set this value to 0 (disabled) as pods in a StatefulSet are usually parts of clusters and should be allowed enough period to do a clean shutdown.

The containers part defines the image that this container uses (Apache httpd:2.4 in our case) and the port that the container will listen on (80).

The volumeMounts part is where you define the Persistent Volume Claim. The name of the volume claim is essential here, and it must match the VolumeClaimTemplate. It also represents the mount point that will be used inside the pod to access the persistent disk (/var/www/html).

```
  volumeClaimTemplates:
  - metadata:
```

```
    name: www
    annotations:
    volume.beta.kubernetes.io/storage-class: www-disk
    spec:
    accessModes: [ “ReadWriteOnce” ]
    resources:
    requests:
         storage: 1Gi
```

The VolumeClaimTemplate, in turn, connects to the Storage Class (defined earlier) specifying instructions, such as the access mode and the requested storage (1 GB).

6.4 Creating The StatefulSet

Now that we have the definition file in place, we can use kubectl to apply it as follows:

```
kubectl apply -f apache-stateful.yaml
```

Since the definition file contains a StorageClass and a StatefulSet resource, the following output is displayed:

```
storageclass.storage.k8s.io/www-disk created
statefulset.apps/webapp created
```

We see that our resources are available. Let’s see whether or not we have pods:

```
kubectl get pods
NAME       READY   STATUS              RESTARTS   AGE
webapp-0   0/1   ContainerCreating   0      8s
```

You may notice two things here: (1) there is only one pod created while we asked for three, and (2) the pod name contains the StatefulSet name.

This is the expected behavior - the StatefulSet will not create all of the pods at once, much like a Deployment. It maintains order when starting, and stopping the pods. Since StatefulSets maintain the pod identity, the pod name is the StatefulSet name followed by an incremental number.

Now, wait a few seconds and issue kubectl get pods again, you should see an output similar to the following:

```
NAME       READY   STATUS              RESTARTS   AGE
webapp-0   1/1   Running          0          43s
webapp-1   0/1   ContainerCreating   0      11s
```

Later on, the output becomes:

```
NAME       READY   STATUS   RESTARTS   AGE
webapp-0   1/1   Running   0          112m
webapp-1   1/1   Running   0          111m
webapp-2   1/1   Running   0          111m
```

The outcome: all of our pods are now started.

6.5 Creating A Headless Service For Our StatefulSet

Right now, the pods are running, but how can a web server access another one? This is done through the Service, so now we need to create one. Open a new YAML file called apache_statefulset_service.yaml and add the following to it:

```
apiVersion: v1
kind: Service
metadata:
  name: web-svc
  labels:
     app: web
spec:
  ports:
  - port: 80
     name: web
  clusterIP: None
  selector:
     app: web
```

Don't forget to specify the service name and the pod selector labels to match the ones that you defined in the StatefulSet definition file.

Create the service by using kubectl:

```
kubectl apply -f apache_statefulset_service.yaml
service/web-access-svc created
```

6.6 Listing The Created Components

Let's have a look at the created components:

```
$ kubectl get statefulset
NAME  READY   AGE
webapp   3/3     21h
```

The StatefulSet was created with the required number of pods.

```
$ kubectl get pv
NAME                                       CAPACITY   ACCESS MODES
RECLAIM POLICY   STATUS   CLAIM                  STORAGECLASS   REASON
AGE
pvc-077a1891-a25b-11e9-9ecf-42010a800184   1Gi        RWO
Delete           Bound default/www-webapp-2   www-disk             21h
pvc-e79d8843-a25a-11e9-9ecf-42010a800184   1Gi        RWO
Delete           Bound default/www-webapp-0   www-disk             21h
pvc-fa398e2a-a25a-11e9-9ecf-42010a800184   1Gi        RWO
Delete           Bound default/www-webapp-1   www-disk             21h
```

We have the Persistent Volumes...

```
$ kubectl get pvc
NAME             STATUS   VOLUME
CAPACITY   ACCESS MODES   STORAGECLASS   AGE
www-webapp-0   Bound  pvc-e79d8843-a25a-11e9-9ecf-42010a800184   1Gi
RWO            www-disk        21h
www-webapp-1   Bound  pvc-fa398e2a-a25a-11e9-9ecf-42010a800184   1Gi
RWO            www-disk        21h
www-webapp-2   Bound  pvc-077a1891-a25b-11e9-9ecf-42010a800184   1Gi
RWO            www-disk        21h
```

...and the Persistent Volume Claims. Let's see how we can connect and use our pods.

6.7 Connecting One Pod To Another Through The Headless Service

We need to test our setup. Let's open a bash shell to one of the pods:

```
kubectl exec -it webapp-0 bash
```

The httpd image isn't shipped with curl by default, so we need to install it:

```
apt update && apt install curl
```

Once it's installed, we can try connecting to the Service:

```
root@webapp-0:/usr/local/apache2# curl web-svc
<html><body><h1>It works!</h1></body></html>
```

This is the default page that Apache displays. The Service is routing the request to the backend pods.

The StatefulSet is all about uniquely identifying pods. So, let's try connecting to a specific pod:

```
root@webapp-0:/usr/local/apache2# curl webapp-1.web-svc
<html><body><h1>It works!</h1></body></html>
```

By prefixing the service name to the pod name, you can connect to that specific pod.

6.8 Deleting The StatefulSet

We start by deleting the Headless Service:

```
kubectl delete -f apache_stateful_service.yaml
service "web-svc" deleted
```

We could achieve the same result by running kubectl delete service web-svc.

To delete the StatefulSet with the Persistent Volume and the Persistent Volume Claims, we use the definition file:

```
kubectl delete -f apache_stateful.yaml
storageclass.storage.k8s.io "www-disk" deleted
statefulset.apps "webapp" deleted
```

The controller honors the ten second grace time and gives the pods time to clean up. In our example, Apache shouldn't take more than a few milliseconds to shut down. But, if it were serving thousands of requests, it would take more time to terminate.

Summary

- A StatefulSet is another Kubernetes controller that manages pods just like Deployments. But it differs from a Deployment in that it is more suited for stateful apps.
- A stateful application requires pods with a unique identity (for example, hostname).

One pod should be able to reach other pods with well-defined names.

- For a StatefulSet to work, it needs a Headless Service. A Headless Service does not have an IP address. Internally, it creates the necessary endpoints to expose pods with DNS names. The StatefulSet definition includes a reference to the Headless Service, but you have to create it separately.
- By nature, a StatefulSet needs persistent storage so that the hosted application saves its state and data across restarts. Kubernetes provides Storage Classes, Persistent Volumes, and Persistent Volume Claims to provide an abstraction layer above the cloud provider's storage-offering mechanism.
- Once the StatefulSet and the Headless Service are created, a pod can access another one by name prefixed with the service name.

7. Kubernetes Resource Requests And Limits 101

The core function of Kubernetes is resource scheduling. Kubernetes scheduler makes sure that containers get enough resources to execute properly. This process is governed by the scheduling policies. Before digging into how the scheduler works, let's make sure we understand the basic constructs of resource definitions, allocation, and restrictions inside the Kubernetes cluster.

7.1 Resources Types

Kubernetes has only two built-in manageable resources: ***CPU*** and ***memory***. CPU base units are cores, and memory is specified in bytes. These two resources play a critical role in how the scheduler allocates pods to nodes. Memory and CPU can be requested, allocated, and consumed. You should always set the right CPU memory values - this way you will be in total control of your cluster and ensure that a misbehaving application does not impact capacity available for other pods in your cluster.

7.2 Requests & Limits

Kubernetes uses the requests & limits structure to control resources such as CPU and memory.

- ***Requests*** are what the container is guaranteed to get. For example, If a container requests a resource, Kubernetes will only schedule it on a node that can give it that resource.

- ***Limits,*** on the other hand, is the resource threshold a container can never exceed. The container is only allowed to go up to the limit, and then it is restricted.

CPU is a **compressible resource**, which means that once your container reaches the limit, it will keep running but the operating system will throttle it, and continuously de-schedule CPU use. Memory, on the other hand, is a **non-compressible resource**. Once your container reaches the memory limit, it will be terminated, aka OOM (Out of Memory) killed. If your container repeatedly gets OOM killed, Kubernetes will report that it's in a crash loop.

Note that the limit can never be lower than the request. Kubernetes will throw an error and won't let you run the container if your limit is higher than the request.

TIP: Set requests and limits at the container level. It's a good practice to set it at the container level for more control and a more efficient distribution of containers. If your main container consumes gigabytes of memory and a sidecar container needs a few megabytes, setting the request and limits at the pod level will give them the same amount of memory. That's a lot of waste. If you are using Magalix Autopilot for Kubernetes, it will automatically set and update the values of limits and requests. .

7.3 Defining CPU Requests & Limits

To specify a CPU request for a Container, include the **resources:requests** field in the Container's resource manifest. Similarly, to specify a CPU limit, include **resources:limits**.

CPU Units At Different Cloud Providers

CPU unit inside Kubernetes is originally equivalent to one hyperthread if you are running on a bare-metal Intel process with Hyperthreading. It's important to understand how this is mapped to different CPU capacities of the major cloud providers out there. Misinterpreting these may cause bad performance results inside your Kubernetes cluster. The table below is a quick mapping of different cloud providers' CPU units.

Infrastructure	1 CPU Equivalent
AWS	1 vCPU
Azure	1 vCore
GCP	Core
IBM	1 vCPU
bare-metal Intel processor with Hyperthreading	1 Hyperthreading

7.4 Define Memory Requests & Limits

To specify a memory request for a Container, include the resources requests field in the Container's resource manifest. To specify a memory limit, include

resources:limits. The Container has a memory request of 100 MiB and a memory limit of 200 MiB. If you want to learn more about Memory requests & limits see this reference from the Kubernetes documentation.

Meaning Of Memory

Limits and requests for memory are measured in bytes. You can express memory as a plain integer or as a fixed-point integer using one of these suffixes: E, P, T, G, M, K.

Take a look at the below example. It is a Pod with two containers. Each Container has a request of 0.5 CPU and 300MiB of memory. Each Container has a limit of 1 CPU and 500MiB of memory.

```
apiVersion: v1
kind: Pod
metadata:
  name: magalix-agent
spec:
  containers:
  - name: database
    image: mongodb
    resources:
      requests:
        memory: "300Mi"
        cpu: "500m"
      limits:
        memory: "500Mi"
        cpu: "1000m"
  - name: magalix-agent
    image: magalix-agent
    resources:
      requests:
        memory: "300Mi"
        cpu: "250m"
      limits:
        memory: "500Mi"
        cpu: "1000m"
```

7.5 Namespace Settings

Ideally, you want your team members to always set limits and resources, but in the real world, sometimes your team might forget to do so. Engineers can easily forget to set the resources, or someone may just get into the habit of overprovisioning to be on the safe side.

You can easily prevent these scenarios and set up ***ResourceQuotas*** and ***LimitRanges*** at the Namespace level.

ResourceQuotas

You can lock namespaces using ResourceQuotas. ResourceQuotas let you look at how you can restrict CPU and Memory resource usage for containers inside the namespace. A Quota for resources might look something like this:

```
apiVersion: v1
kind: ResourceQuote
metadata:
         name: demo
spec:
     hard:
          requests.cpu: 500m
          requests.memory: 100Mib
          limits.cpu: 700m
          limits.memory: 500Mib
```

You can see there are four specific sections. However, configuring each of these sections is optional.

- **requests.**cpu is the maximum combined CPU requests in millicores for all the containers in the Namespace. You can have as many containers as you want as long as the total CPU requested in the Namespace is less than 500m.
- **requests.memory** is the maximum amount of combined Memory requests for all the containers in the Namespace. You can have as many containers as you want use this quota as long as the total requested Memory in the Namespace is less than 100MiB.
- **limits.cpu** is the maximum combined CPU limits for all the containers in the Namespace.
- **limits.memory** is the maximum combined Memory limits for all containers in the Namespace.

TIP: If you are using a production and development Namespace, it's recommended that you avoid defining quota on the production Namespace, and define strict quotas on the development Namespace. You don't want your production containers to be throttled or evicted because the dev env needs more resources!

7.6 LimitRange

The LimitRange applies to an individual container and can help prevent your team

members from creating super tiny or super large containers inside the Namespace. A LimitRange might look like this:

```
apiVersion: v1
kind: limitrange
metadata:
       name: demo
spec:
       limits:
     - default:
             cpu: 600m
             memory: 100Mib
      defaultRequest:
             cpu: 100m
             memory: 50Mib
       max:
             cpu: 100m
             memory: 200Mib
        min:
             cpu: 10m
             memory: 10Mib
        type: Container
```

You can see there are four optional sections.

- The **default section** sets up the default limits for a container in a pod. Containers that don't explicitly set these themselves will be assigned default values.
- The **defaultRequest** section sets up the default requests for a container in a pod. containers that don't explicitly set these themselves will get assigned the default values will get assigned these values by default.
- The **max section** will set up the maximum limits that a container in a Pod can set. The default section cannot be higher than this value. Also, limits set on a container cannot be higher than this value. Containers that don't explicitly set these values will be assigned the max values as the limit if this value is set and the default section is not.
- The **min section** sets up the minimum Requests that a container in a Pod can set. The **defaultRequest** section cannot be lower than this value. Also, requests set on a container cannot be lower than this value either. The min value becomes the **defaultRequest** value too if this value is set and the **defaultRequest** section is not.

7.7 The Lifecycle Of A Kubernetes Pod

It's important to understand how pods are granted resources, and what happens when they exceed what is allocated (or the overall capacity of your cluster) so that you can tune your containers and cluster capacity correctly. Below is the typical pod scheduling workflow:

1. You allocate resources to your pod through the spec and run kubectl apply command.
2. Kubernetes scheduler will use a round-robin scheduling policy to pick a Node to run your pod.
    a. For each node, Kubernetes checks if the node has enough resources to fulfill the resource requests.
2. The pod is scheduled for the first node that has enough resources.
3. The pod goes into a pending state if none of the available nodes have enough resources.
    a. If you are using CA autoscaler, your cluster will scale the number of nodes to allocate more capacity.
5. If a node is running pods where the sum of their limits is more than its available capacity, Kubernetes goes into an Overcommitted State.
    a. Because CPU can be compressed, Kubernetes will make sure your containers get the CPU they requested and will throttle the rest.
    b. Memory cannot be compressed, so Kubernetes needs to start making decisions on what containers to terminate if the Node runs out of memory..

7.8 What If Kubernetes Goes Into Overcommitted State?

Keep in mind that Kubernetes optimizes the whole system's health and availability. When it goes into the overcommitted state, Kubernetes scheduler may make the decision to kill pods. Generally, if a pod is using more resources than requested, that pod becomes a candidate for termination. Here are some conditions that you should keep in mind:

- A pod does not have a request for CPU and memory defined. Kubernetes assumes this pod is using more than requested by default.
- A pod has a request defined but is using more than the requested but still under their limit.
- If multiple pods have exceeded their requests, Kubernetes will rank them based on their priority - the lowest priority pods will be terminated first.
- If all pods have the same priority, Kubernetes starts with the pod that is exceeding its requested resources the most.
- Kubernetes may kill pods within their request limits if a critical component (such as kubelet or docker engine) is using more resources than what's allocated. However, these are rare occurrences.

7.9 Final Thoughts

- Setting the request for CPU or memory higher than any of the cluster nodes can handle will place the container in a pending state indefinitely, until a large enough node is available.
- Setting a single node limit higher than its capacity to handle puts your node(s) at risk of being inaccessible due to CPU overload - unfortunately, high limits allow the container(s) to monopolize all the resources in the node. Other nodes may fail and you may not be able to ssh into the node to kill those containers and remedy the situation.
- Setting the CPU limit to a very small value may result in heavy throttling of your container. It's highly recommended that you monitor CPU throttling metrics, specifically pu.cfs_period_us and the cpu.cfs_quota_us

8. Jobs, The Task Pods

8.1 Kubernetes Jobs Use Cases

Kubernetes features several controllers for managing pods. We have ReplicaSets, DaemonSets, StatefulSets, and Deployments. Each one has its own scenario and use case. However, they all share one common property - they ensure that their pods are always running. If a pod fails, the controller restarts it, or reschedules it, to another node to make sure the application the pods are hosting is running.

So, what if we do want the pod to terminate? There are many scenarios when you don't want a process to keep running indefinitely. Think of a log rotation command - log rotation is the process of archiving (compressing) logs files that are older than a particular time threshold and deleting the really ancient ones. We definitely don't need a process like that to be running continuously. Instead, it gets executed, and once the task is complete, it returns the appropriate exit status that reports whether the result is a success or failure.

Kubernetes Jobs ensure that one or more pods execute their commands and exit successfully. When all the pods have exited without errors, the Job gets completed. When the Job gets deleted, any created pods get deleted as well.

8.2 Your First Kubernetes Job

Creating a Kubernetes Job, like other Kubernetes resources, is done through a definition file. Open a new file; you can name it my_job.yaml. Then, add the following content to the file:

```
apiVersion: batch/v1
kind: Job
metadata:
 name: say-something
spec:
 template:
   metadata:
     name: say-something
   spec:
     containers:
     - name: say-something
```

```
      image: busybox
      command: ["echo", "Running a job"]
    restartPolicy: OnFailure
```

As with other Kubernetes resources, we can apply this definition to a running Kubernetes cluster using kubectl as follows:

```
$ kubectl apply -f my_job.yaml
job.batch/say-something created
```

Let's see what pods were created for us:

```
$ kubectl get pods
NAME                  READY   STATUS              RESTARTS   AGE
say-something-fqjfd   0/1   ContainerCreating   0          2s
```

Give it a few seconds and then run the same command again:

```
$ kubectl get pods
NAME                  READY   STATUS   RESTARTS   AGE
say-something-fqjfd   0/1   Completed   0          9s
```

The pod status is not running; it is "Completed," as the job ran and exited successfully. The job we've just defined had an effortless task: echo "Running a job" to the standard output.

Before moving any further, let's ensure that the job indeed did what we instructed it to do:

```
$ kubectl logs say-something-fqjfd
Running a job
```

The logs show that this pod echoed "Running a job." Therefore, the job was successful.

8.3 The Kubernetes Job Definition File

This definition starts with the apiVersion, kind, and metadata, similar to the other Kubernetes config. The spec part contains the pod template and looks precisely like a pod definition without the kind and apiVersion fields. In our example, we are basing our container on the busybox image. We're instructing it to execute a command that prints "Running a job."

Kubernetes Job RestartPolicy

The restartPolicy cannot be set to always. By definition, a Job should not restart a pod when it terminates successfully. Thus, options available for restartPolicy are “Never” and “OnFailure.”

Kubernetes Job Needs No Pod Selector

Notice that we didn’t specify a pod selector like in other pod controllers (Deployments, ReplicaSets, etc.).

A Job does not need a pod selector because the controller automatically creates a label for its pods. It ensures that this label is not in use by other jobs or controllers, and it uses it to match and manage its pods.

8.4 Jobs Completions And Parallelism

So far, we’ve seen how we can run one task defined inside a Job object, more commonly known as the “run-once” pattern. However, real-world scenarios involve other, more diverse, patterns as well.

Multiple Single Jobs

For example, we may have a queue of messages that needs processing. We must spawn consumer jobs that pull messages from the queue until it’s empty. To implement this pattern in Kubernetes Jobs, we set the .spec.completions parameter to a number (must be a non-zero, positive number). The Job starts spawning pods up until it reaches the completion number. The Job regards itself as complete when all the pods terminate with a successful exit code. Let’s check out another example - modify our definition file to look as follows:

```
apiVersion: batch/v1
kind: Job
metadata:
name: consumer
spec:
```

```
template:
  metadata:
    name: consumer
  spec:
    containers:
    - name: consumer
      image: busybox
      command: ["/bin/sh","-c"]
      args: ["echo 'consuming a message'; sleep 5"]
    restartPolicy: OnFailure
```

This definition is very similar to the one we used before, with a few differences:

- We specify the completions parameter to be 5.
- We change the command that the container inside the pod uses to include a five-second delay, which ensures that we can see the pods created and terminated.

Issue the following command:

```
kubectl apply -f my_job.yaml && kubectl get pods --watch
```

This command applies the new definition file to the cluster and immediately starts displaying the pods and their status. The --watch flag saves us from having to type the command over and over as it automatically displays any changes in the pod statuses:

```
job.batch/consumer created
NAME             READY   STATUS              RESTARTS   AGE
consumer-kwwxs   0/1     ContainerCreating   0          0s
consumer-kwwxs   1/1     Running             0          2s
consumer-kwwxs   0/1     Completed           0          7s
consumer-xvb2h   0/1     Pending             0          0s
consumer-xvb2h   0/1     Pending             0          0s
consumer-xvb2h   0/1     ContainerCreating   0          0s
consumer-xvb2h   1/1     Running             0          2s
consumer-xvb2h   0/1     Completed           0          7s
consumer-g58l5   0/1     Pending             0          0s
consumer-g58l5   0/1     Pending             0          0s
consumer-g58l5   0/1     ContainerCreating   0          0s
consumer-g58l5   1/1     Running             0          2s
consumer-g58l5   0/1     Completed           0          7s
consumer-595bl   0/1     Pending             0          0s
consumer-595bl   0/1     Pending             0          0s
consumer-595bl   0/1     ContainerCreating   0          0s
consumer-595bl   1/1     Running             0          2s
consumer-595bl   0/1     Completed           0          7s
```

```
consumer-whtmp   0/1     Pending             0          0s
consumer-whtmp   0/1     Pending             0          0s
consumer-whtmp   0/1     ContainerCreating   0          0s
consumer-whtmp   1/1     Running             0          2s
consumer-whtmp   0/1     Completed           0          7s
```

As you can see from the above output, the Job created the first pod. When the pod terminated without failure, the Job then spawned each one consecutively until the last of the ten pods were created, and terminated, with no failure.

Multiple Parallel Jobs (Work Queue)

Another pattern may involve the need to run multiple jobs, but instead of running them one after another, we need to run several of them in parallel. Parallel processing decreases the overall execution time. It has its application in many domains, like data science and AI.

Modify the definition file to look as follows:

```
apiVersion: batch/v1
kind: Job
metadata:
name: consumer
spec:
parallelism: 5
template:
  metadata:
    name: consumer
  spec:
    containers:
    - name: consumer
      image: busybox
      command: ["/bin/sh","-c"]
      args: ["echo 'consuming a message'; sleep $(shuf -i 5-10 -n 1)"]
    restartPolicy: OnFailure
```

Here we didn't set the .spec.completions parameter - instead, we specified the parallelism one. The completions parameter (in our case) defaults to parallelism (5). The Job now has the following behavior: five pods will get launched at the same time; all of them are executing the same Job. When one of the pods terminates successfully, this means that the whole Job is done. No more pods get spawned, and the Job eventually terminates. Let's apply this definition:

```
$ kubectl apply -f my_job.yaml && kubectl get pods --watch
job.batch/consumer created
NAME                              READY   STATUS   RESTARTS   AGE
consumer-q99zs                    0/1   Pending   0          0s
consumer-9k6fs                    0/1   Pending   0          0s
consumer-5htz9                    0/1   Pending   0          0s
consumer-9v6l6                    0/1   Pending   0          0s
consumer-sb6wp                    0/1   Pending   0          0s
consumer-9k6fs                    0/1   Pending   0          1s
consumer-5htz9                    0/1   Pending   0          1s
consumer-sb6wp                    0/1   Pending   0          1s
consumer-9v6l6                    0/1   Pending   0          1s
consumer-q99zs                    0/1   Pending   0          1s
consumer-9k6fs                    0/1   ContainerCreating   0       1s
consumer-5htz9                    0/1   ContainerCreating   0       1s
consumer-sb6wp                    0/1   ContainerCreating   0       1s
consumer-9v6l6                    0/1   ContainerCreating   0       1s
consumer-q99zs                    0/1   ContainerCreating   0       1s
consumer-q99zs                    1/1   Running          0          11s
consumer-9k6fs                    1/1   Running          0          14s
consumer-sb6wp                    1/1   Running          0          17s
consumer-q99zs                    0/1   Completed        0          19s
consumer-9k6fs                    0/1   Completed        0          19s
consumer-9v6l6                    1/1   Running          0          21s
consumer-5htz9                    1/1   Running          0          25s
consumer-sb6wp                    0/1   Completed        0          25s
consumer-9v6l6                    0/1   Completed        0          27s
consumer-5htz9                    0/1   Completed        0          33s
```

In this scenario, the Kubernetes Job is spawning five pods at the same time. It is the responsibility of the pods to know whether or not their peers have finished. In our example, we assume that we are consuming messages from a message queue (like RabbitMQ). When there are no more messages to consume, the job receives a notification that it should exit. Once the first pod exits successfully:

- No more pods are spawned.
- Existing pods finish their work and exit as well.

In the above example, we changed the command that the pod executes to make it sleep for a random number of seconds (from five to ten) before it terminates. This way, we are roughly simulating how multiple pods can work together on an external data source like a message queue or an API.

8.5 Kubernetes Job Failure And Concurrency Considerations

A process running through a Kubernetes Job is different than a daemon. A daemon represents some service with a web interface (for example, an API). Traditionally, a daemon is programmed to be stopped or restarted without issues. On the other hand, a process that is designed to run once may run into errors when it exits prematurely. Kubernetes Jobs restart the container inside the pod if it fails. It may also restart the whole pod, or reschedule it to another node, for multiple reasons. For example:

- The node has crashed.
- The node was rebooted or upgraded.
- The pod was consuming more resources than the node's capacity..

Even if you set .spec.parallelism = 1, .spec.completions = 1 and .spec.template.spec.restartPolicy = "Never," there is no guarantee that the Kubernetes Job will run the process more than once.

The bottom line is: the process that runs through a Job must be able to handle premature exit (lock files, cached data, etc.), and it must also be capable of surviving while multiple instances are running.

The Pod Failure Limit

If you set the restartPolicy = "OnFailure" and your Pod has a problem that makes it repeatedly fail (a configuration error, an unreachable database, API, etc.), does that mean that the Kubernetes Job will keep restarting the Pod indefinitely? Fortunately, no. A Kubernetes Job will retry running a failed pod every time interval - this time interval starts at ten seconds then doubles, i.e. 10, 20, 40, 80, etc. However, once six minutes have passed, the Job will no longer restart the failing Pod.

You can override this behavior by setting the spec.backoffLimit to the number of times the Kubernetes Job should restart the failing Pod.

Note that setting restartPolicy = "OnFailure" terminates the container when the backoff limit is reached. As such, it may be more challenging when you need to

track down what caused the Job to fail. In such a case, you may want to set the restartPolicy = “Never” and start debugging the problem.

8.6 Limiting The Kubernetes Job Execution Time

Sometimes, you’re more interested in running your Job for a specific amount of time regardless of whether the process completes successfully. Think of an AI application that needs to consume data from Twitter and you’re using a cloud instance where the provider charges you for the amount of CPU and network resources you are utilizing. You’re using a Kubernetes Job for data consumption, and in this case, you don’t want it to run for more than one hour.

Kubernetes Jobs offer the spec.activeDeadlineSeconds parameter and setting this parameter to a specific number will terminate the Job immediately once your specified number of seconds is reached.

Notice that this setting overrides .spec.backoffLimit, which means that if the pod fails and the Job reaches its deadline limit, it will not restart the failing pod - it will stop immediately.

In the following example, we are creating a Job that has both a backoff limit and a deadline:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: twitter-consumer
spec:
  backoffLimit: 5
  activeDeadlineSeconds: 20
  template:
      spec:
        containers:
        - name: consumer
          image: busybox
          command: ["/bin/sh", "-c"]
          args: ["echo 'Consuming data'; sleep 1; exit 1"]
        restartPolicy: OnFailure
```

In this definition, we are instructing the container to echo some text, sleep for one

second, and then exit with a non-zero status (unsuccessful exit). Notice that we intentionally set the deadline to be twenty seconds while the *backofflimit* is 5. Let's see what happens when we deploy this job to the cluster:

```
$ kubectl apply -f my_job.yaml && kubectl get pods --watch
job.batch/twitter-consumer created
NAME                    READY   STATUS              RESTARTS   AGE
twitter-consumer-kfvrj   0/1     ContainerCreating   0          0s
NAME                    READY   STATUS              RESTARTS   AGE
twitter-consumer-kfvrj   1/1     Running             0          4s
twitter-consumer-kfvrj   0/1     Error               0          6s
twitter-consumer-kfvrj   1/1     Running             1          10s
twitter-consumer-kfvrj   0/1     Error               1          11s
twitter-consumer-kfvrj   0/1     Terminating         1          20s
twitter-consumer-kfvrj   0/1     Terminating         1          20s
```

As you can see from the above output, the Kubernetes Job started the pod, but since the pod shortly fails, it restarts. It keeps on restarting the failing pod, but once the activeDeadlineSeconds parameter is reached (twenty seconds), the job exits immediately (despite restarting the pod only twice and not reaching the backoffLimit of five).

8.7 Kubernetes Job Deletion And Cleanup

When a Kubernetes Job finishes, neither the Job nor the pods that it created will be deleted automatically - you will have to remove them manually. This feature ensures that you're still able to view the logs, the status of the finished Job, and its pods.

A job can be deleted by using kubectl as follows:

```
kubectl delete jobs job_name
```

The above command deletes the specified Job and all its child pods. Like other Kubernetes controllers, you can opt to remove only the Job, leaving its pods bypassing the cascade=false flag. For example:

```
kubectl delete jobs job_name cascade=false
```

It's worth noting that there is a new feature in Kubernetes that allows you to specify the number of seconds after which a completed Job gets deleted (together with its pods). This feature uses the TTL controller. Note that this feature is still in alpha state - to use it, we can modify our definition file to look as follows:

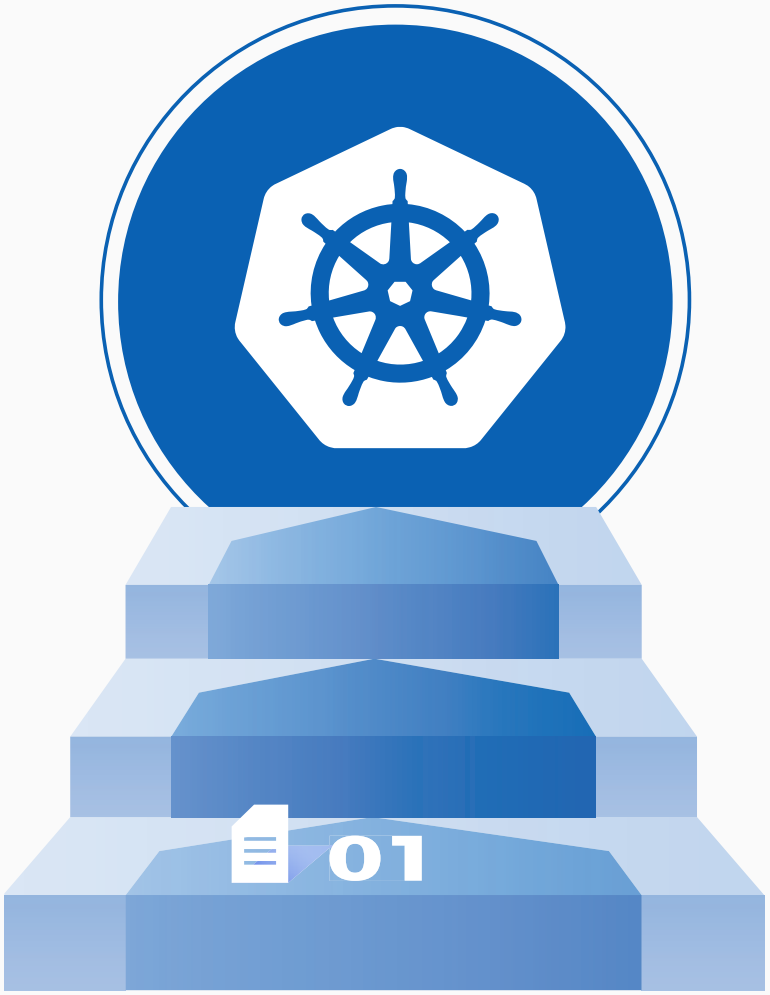
```
apiVersion: batch/v1
kind: Job
metadata:
  name: twitter-consumer
spec:
  backoffLimit: 5
  activeDeadlineSeconds: 20
  ttlSecondsAfterFinished: 60
  template:
      spec:
      containers:
      - name: consumer
        image: busybox
        command: ["/bin/sh", "-c"]
        args: ["echo 'Consuming data'; sleep 1; exit 1"]
      restartPolicy: OnFailure
```

This new definition will make sure the finished Job objects will be deleted (with their pods) one minute after their completion.

Summary

- Kubernetes Jobs are used when you want to create pods that will do a specific task and then exit.
- Kubernetes Jobs do not need pod selectors by default (as pods would); their labels and selectors are handled automatically by the Job.
- The restartPolicy for a Job accepts “Never” and “OnFailure”
- Jobs use completion and parallelism parameters to control the patterns the pods run through. Job pods can run as a single task, several sequential tasks, or some parallel tasks in which the first task that finishes instructs the subsequent pods to complete and exit.
- You can control how many times a Job attempts to restart a failing pod using the .spec.backoffLimit. Note: this limit defaults to six.
- You can control how long a job will run using the .spec.activeDeadlineSeconds. This limit overrules the backoffLimit - therefore the Job does not attempt to restart a failing pod if the deadline is reached.
- Jobs and their pods do not get deleted automatically when they finish. You will have to manually delete them, or use the ttlSecondsAfterFinished controller, which is still in the alpha stage at the time of this publication.

Conclusion



Congratulations! You've just had your first steps to the amazing world of Kubernetes. Let's quickly recap what we've discussed in this book.

We started by an introduction to the technology, what it is, what it tries to solve and its basic architecture. Then we moved on to discussing Pods. a Pod is the atomic unit of deployment for Kubernetes. A Pod can host one or more containers. If that's the case, multiple containers in the same pod are treated as one unit: they are deployed together, moved from one node to another together, and destroyed together.

For Kubernetes to maintain a specific number of pod replicas and also ensure that crashing pods get restarted, it uses a higher-level controller called the ReplicaSet. A ReplicaSet is responsible for creating, restarting, and destroying pods while making sure that there is a well-defined number of pods running at all times. It uses labels to select pods that should under its control. Since we are hosting multiple pods through ReplicaSets, there should be a single point of entry for multiple pods, namely, a load-balancer of some sort. That's the role of Services.

A Service provides an interface through which the client can reach the backend pods. A Service can have multiple types depending on how you intend to use it. For example, we have ClusterIP, NodePort, and LoadBalancer. Each of which has its specific use case.

More often than not, you need to make changes to the application running on Kubernetes. For example, you want to introduce a new feature or a bug fix. The

typical way of doing this is by changing the image name of the application. However, in today's microservices world, you cannot tolerate any downtime. For that reason, Kubernetes introduces the Deployment controller. The Deployment is actually a wrapper around the ReplicaSet. But it adds some functionality to ReplicaSets. For example, you can deploy a new version of the application by just changing the image name of an existing deployment. The controller will ensure that you don't have an application outage by creating new pods at the same time as terminating old ones. A new pod does not enter the service unless the old one is completely terminated. Using Deployments, you can make use of many deployment strategies like rolling updates, A/B testing, Blue/Green deployments, and Canary deployments.

Deployments are great for stateless applications. By stateless, we mean those applications that do not need to maintain their state upon restarts. A typical example of that kind of apps is web servers. If you restart Nginx several times, each time it would work regardless of which requests it was serving before it died. It does not need a parent service to be up beforehand. If you are running multiple instances of the webserver behind a load balancer, increasing or decreasing the number of replicas should not need any special handling. In contrast, stateful applications do maintain their state. For example, if you are hosting a database cluster, each instance needs to have a well-defined hostname, network address, and startup order. If the cluster needs a controller like ZooKeeper, all the instances need to be aware of the leader node before they can function correctly. For those types of applications, Deployments are not the suitable choice. Kubernetes introduced the StatefulSets for that purpose. Using a StatefulSet, you guarantee that each replica of the application has a hostname that does not change upon restart. Each instance has storage attached to it. When the instance gets restarted, it finds the same storage that it was using before it died. Additionally, SatefulSets pods are always started one by one and the pods are named in sequential order. SatetefulSets are used by stateful applications like databases and message queueing systems.

Finally, sometimes you have a business requirement that does not require a pod to be up and running all the time. In fact, you require that the pod is not restarted when the process that is running exits. For example, you may have a message queueing system from which you need to pull data. The client application is hosted on a pod, that pod should exit successfully when there are no more messages in the queue. Kubernetes offers the Job controller for this. A Job will ensure that one or more pods are spawned to execute a task. Once the task is completed, the pod is not restarted. You can optionally instruct the Job to restart the pod if the process failed.

I hope by now you have a firm understanding of the various controllers that Kubernetes uses to maintain does its job. You are now a Kubernetes administrator.

Thank You

This book brought to you by Magalix team

Magalix Accelerates your cloud-native journey. make your Kubernatenetes and team production ready in a snap

MAGALIX

team@magalix.com

Magalix.com MagalixCorp