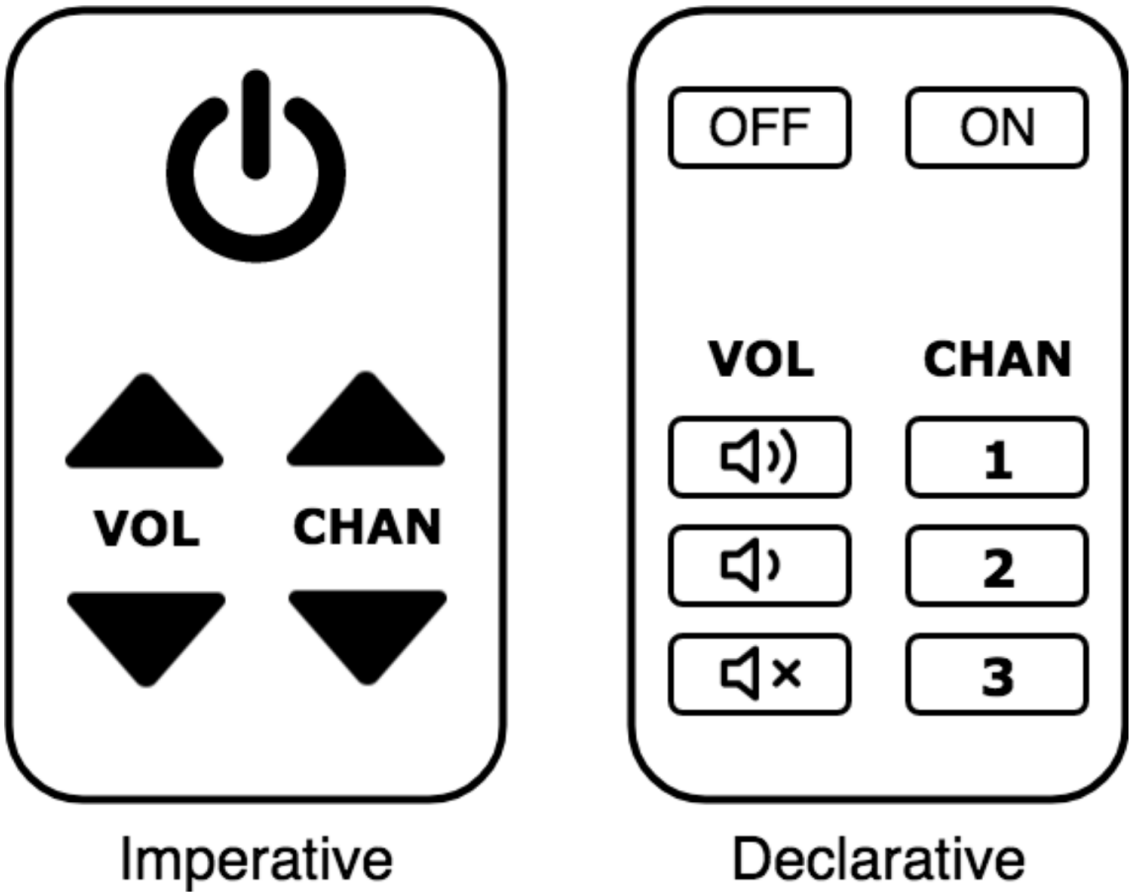
Declarative

One of the more prominent paradigms to emerge from the DevOps movement is the model of declarative systems and configuration. Simply put, with declarative models, you describe what you want to be achieved, as opposed to how to get there. This is in contrast to an imperative model, in which you describe a sequence of instructions to manipulate the state of the system to reach your desired state.

To illustrate this difference, imagine the following two styles of a television remote control: an imperative style and a declarative style. Both remotes can control the TV's power, volume, and channel. For the sake of discussion, assume that the TV has only three volume settings (loud, soft, mute), and only three channels (1, 2, 3).



IMPERATIVE REMOTE EXAMPLE

Suppose you had the simple task of changing to channel 3 using both remotes. To accomplish this task using the imperative remote, you would use the “channel up” button, which signals the TV to increment the current channel by one. To reach channel 3, you would keep pressing the “channel up” button, some number of times until the TV reaches the desired channel.

DECLARATIVE REMOTE EXAMPLE

Contrast this to the declarative remote, which provides individual buttons that jump directly to the specific, numbered channel. In this case, to switch to channel 3, you would press the “channel 3” button once, and the TV would be on the correct channel. With the declarative remote, you are declaring your intended end state (I want the TV tuned to channel 3). Whereas with the imperative remote, you are describing the actions needed to be performed to achieve your desired state (keep pressing “channel up” until the TV is tuned to channel 3).

You may have noticed that in the imperative approach of changing channels, it required some thought on the part of the user about whether or not to continue pushing the “channe up” button, depending on what channel the TV is currently tuned to. Whereas in the declarative approach, you can press the “channel 3” button without a second thought. This is because the “channel 3” button in the declarative remote is considered to be “idempotent,” whereas the “channel up” button in the imperative remote is not.

IDEMPOTENCY

Idempotency is a property of an operation, whereby the operation can be performed any number of times and still produce the same result. In other words, an operation is said to be idempotent if you can perform the operation an arbitrary number of times, and the system is in the same state as it would be if you had performed the operation only once. Idempotency is one of the properties distinguishing declarative systems from imperative systems. Declarative systems are idempotent; imperative systems are not.

In mathematics, some examples of idempotent operations are:

- Multiplying a number by 1
- Adding zero to a number

As we continue to compare the usage of the imperative remote control versus the declarative one, this difference becomes apparent. In the imperative remote, the effect that pushing any of the buttons has on the television depends entirely on the current state of the TV prior to pushing the button. For example, pushing the power button on the imperative remote will have one of two results:

- If the television is currently powered off, it will be turned on
- If the television is currently powered on, it will be turned off

The fact that the power button can have different outcomes depending on the current state of the TV means that the button is not idempotent. If you wanted the TV turned on, you would first need to physically look at the TV to see if it was already on, before deciding if the power button should be pressed or not. If it was on, you would want to avoid pushing the power button altogether, since pressing it would have the opposite effect of your desired outcome.

On the other hand, in the declarative remote, no matter how many times you push the ON button, the result is the same: the television will be powered on. Therefore, the ON button is considered idempotent. Without even looking at the TV, you could blindly press the ON button, and even if the TV was already on, the result would be the same.

It is important to note that there are advantages and disadvantages to imperative versus declarative methodologies. Imperative systems are much simpler to implement and understand. They are simpler because they only provide the basic, low-level constructs to the system. For the same reason, imperative systems tend to be more flexible. By providing just basic primitives to manipulate the system, the caller is given all the building blocks needed to materialize their desired end state.

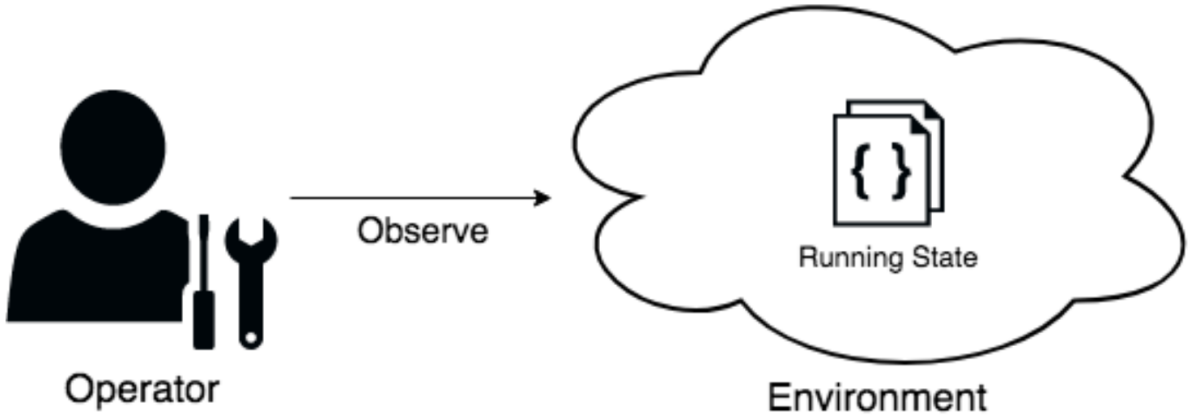
However, when it comes to complex distributed systems, microservices, and dealing with failure scenarios, there are clear advantages to a declarative methodology. In the world of distributed systems, there is much more that can go wrong than traditional monolithic applications that are self-contained on a single host. The following are some common examples of error scenarios one needs to handle when dealing with remote services:

- A request was sent but never received.
- A request was received but never acknowledged.
- A request was acknowledged but never fulfilled.
- As a result of some combination of the above, a request was duplicated.

Development paradigms have shifted towards declarative APIs in order to simplify the interactions between remote services and address these types of error scenarios. If an API is accepting the desired end state, then a client only needs to ensure that the request was acknowledged, and trust that sometime later, the request will eventually be fulfilled.

Observability

Observability is the ability to examine and describe the current running state of a system and be alerted of unexpected conditions. Deployed environments are expected to be observable. In other words, you should always be able to inspect an environment in order to see what is currently running and how things are configured. To that end, service and cloud providers will provide a fair number of methods to promote observability (CLIs, APIs, GUIs, dashboards, alerts & notifications), making it as convenient as possible for users to understand the current state of their environment.

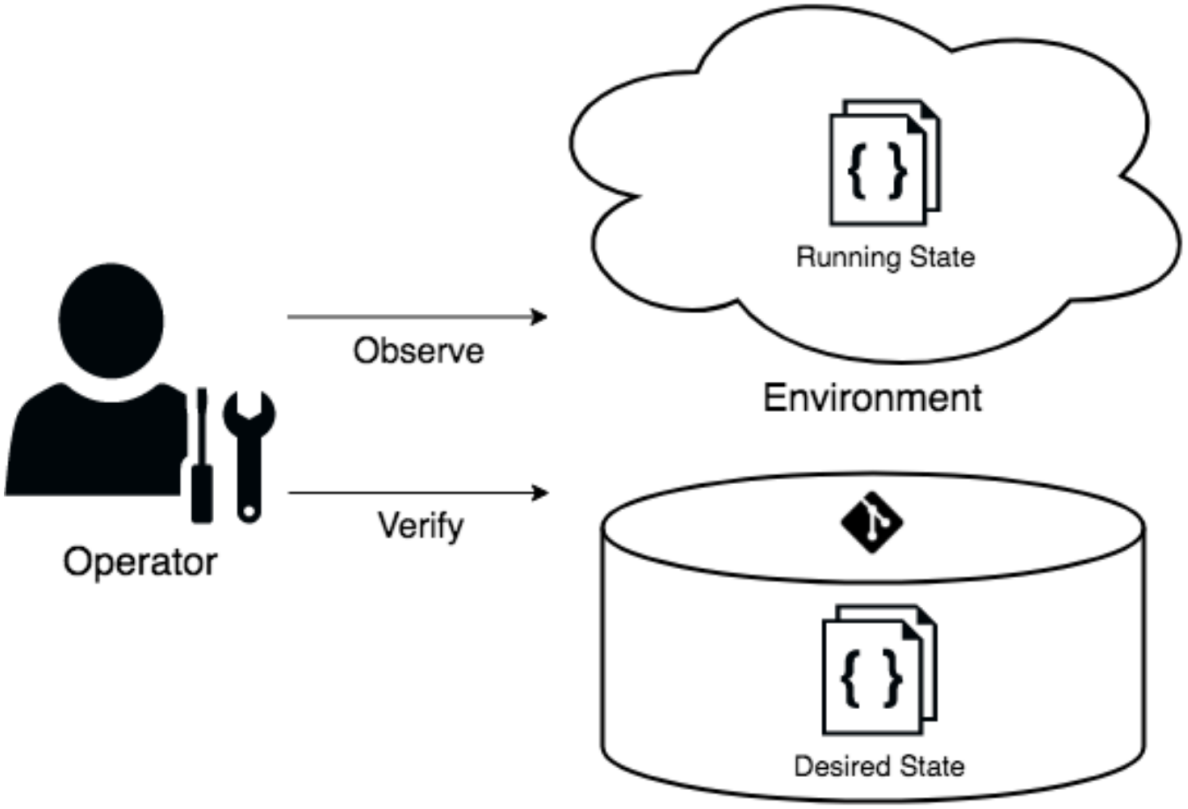


Although these observability mechanisms can help answer the question of “what’s currently running in my environment?” what they cannot answer the question of “for the resources currently configured and running in my environment, are they supposed to be configured and running in that way?” This question can only be answered by the operator and is part and parcel of the duties and responsibilities as an operator.

If you have ever held duties as a system administrator or operator, you’re likely all too familiar with this problem. At one point or another, typically when troubleshooting an environment, you come across a suspect configuration setting, and wonder to yourself: this doesn’t seem right — did someone (possibly yourself) accidentally or mistakenly change this setting?

This problem is really a matter of intention, and the problem statement could alternatively be phrased, “is this system configured the way it was intended to be?” In order to answer that question, there needs to be a “source of truth” or system of record that defines this intention — the way that things should be. This is also described as the “desired state” of the environment and is the foundation of the method of continuous delivery.

In all likelihood, you may already be practicing a fundamental principal, which is to store a copy of your application configuration in source control and use it as a “source of truth” of the desired state of your application. You might not be storing it in Git to drive continuous delivery, but to simply to have a copy of the configuration duplicated somewhere, so that the environment can be reproduced (e.g., in a disaster recovery scenario). This copy, for all intents and purposes, can be thought of as the desired application state, and aside from the disaster recovery use case, it serves another useful purpose. It enables operators, at any point in time, to compare the actual running state to the desired state held in source control, and verify that they are matching.



By storing your desired state into one system (i.e., Git), and regularly comparing that desired state with the running state, you unlock an entirely new dimension of observability. Not only do you have the standard observability mechanisms provided by your provider, but you are now able to detect divergence from the desired state.
Divergence from your desired state, also called configuration drift, can happen for any number of reasons. Common examples include mistakes made by operators, unintended side-effects due to automation, error scenarios. It could even be expected, like a temporary state caused by some transition period (e.g., maintenance mode).

But the most significant reason why there might be a divergence in configuration could be malicious. In the worst case, a bad actor could have compromised the environment and reconfigured the system to run a malicious image. For this reason, observability and

verification are doubly important with regards to security. Without a source of truth of your desired state put in place, and without a mechanism to verify convergence to that source of truth, it is impossible to know that your environment is truly secure.

Rollback

Any good deployment solution will offer some form of a rollback feature. Rollback is the operation of restoring to a previous known good state of the system. It is performed by an operator, typically in response to a problem or regression introduced by the currently deployed state. Rollback is critical to operations since it provides a safety net to operators: the knowledge that they can always fall back to the previous working state, should things go awry with their new deployment.

Deployment systems will provide users with a rollback button to facilitate rollbacks, enabling users to conveniently and quickly return to the previous state. Essentially, they act as a big “undo” command for your service.

Kubernetes

Kubernetes is an open-source container orchestration system released in 2014. OK, but what are containers, and why do you need to orchestrate them?

Containers provide a standard way to package your application's code, configuration, and dependencies into a single resource. This enables developers to ensure that the application will run properly on any other machine regardless of any customized settings that machine may have that could differ from the machine used for writing and testing the code. Docker simplified and popularized containerization, which is now recognized as a fundamental technology used to build distributed systems.

While Docker solved the packaging and isolation problem of individual applications, there was still the question of how to orchestrate the operation of multiple applications into a working distributed system.

- How do containers communicate with each other?
- How is traffic routed between containers?
- How are containers scaled up to handle additional application load?
- How is the underlying infrastructure of the cluster scaled-up to run the required containers?

All these operations are the responsibility of a container orchestration system and are provided by Kubernetes. Kubernetes helps to automate the deployment, scaling, and management of applications using containers.

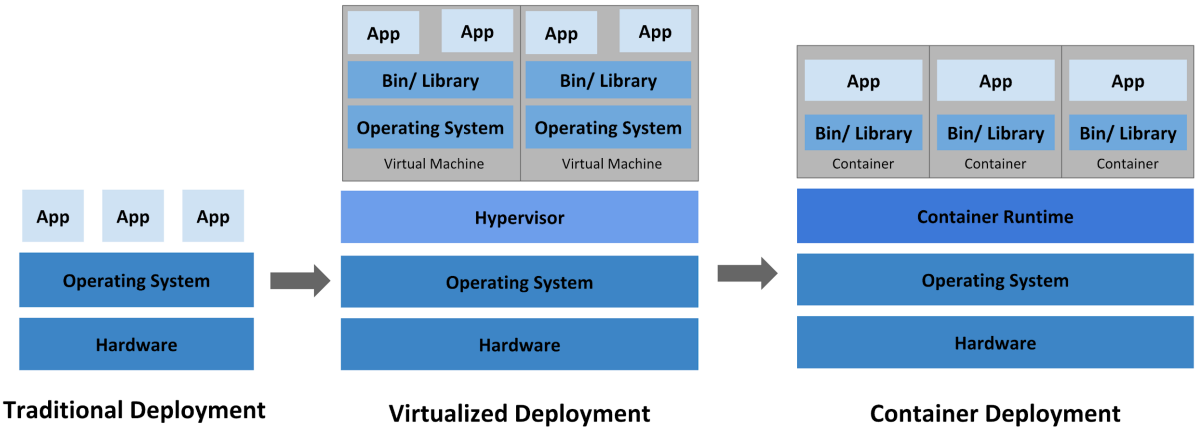
Kubernetes was initially developed and open-sourced by Google based on a decade of experience with container orchestration using Borg, Google's proprietary cluster management system. Because of this, Kubernetes is relatively stable and mature for a system so complex. Because of its open API and extendable architecture, Kubernetes has developed an extensive community around it, which has further fueled its success. It is one of the top GitHub projects (as measured by "stars"), provides great documentation, and has a significant Slack and Stack Overflow community. There is an endless number of blogs and presentations from community members who are sharing their knowledge of how to use Kubernetes. Despite being started by Google, Kubernetes is not influenced by a single vendor. This makes the community open, collaborative, and enables innovation.

Other Container Orchestrators

Since late-2016, Kubernetes has become recognized as the dominant de-facto industry-standard container orchestration system in much the same way as Docker has become the standard for containers. However, there are several major Kubernetes alternatives that address the same container orchestration problem as Kubernetes. Docker Swarm is Docker's native Container Orchestration Engine that was released in 2015. It is tightly integrated with the Docker API and uses a YAML-based deployment model called Docker Compose. Apache Mesos was officially released in 2016 (although it has a history well before then) and supports large clusters, scaling to thousands of nodes.

Going back in time

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



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

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

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

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

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

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

- Agile application creation and deployment: increased ease and efficiency of container image creation compared to VM image use.
- Continuous development, integration, and deployment: provides for reliable and frequent container image build and deployment with quick and easy rollbacks (due to image immutability).
- Dev and Ops separation of concerns: create application container images at build/release time rather than deployment time, thereby decoupling applications from infrastructure.
- Observability not only surfaces OS-level information and metrics, but also application health and other signals.
- Environmental consistency across development, testing, and production: Runs the same on a laptop as it does in the cloud.
- Cloud and OS distribution portability: Runs on Ubuntu, RHEL, CoreOS, on-premises, on major public clouds, and anywhere else.
- Application-centric management: Raises the level of abstraction from running an OS on virtual hardware to running an application on an OS using logical resources.
- Loosely coupled, distributed, elastic, liberated micro-services: applications are broken into smaller, independent pieces and can be deployed and managed dynamically – not a monolithic stack running on one big single-purpose machine.
- Resource isolation: predictable application performance.
- Resource utilization: high efficiency and density.

Why you need Kubernetes and what it can do

Containers are a good way to bundle and run your applications. In a production environment, you need to manage the containers that run the applications and ensure that there is no downtime. For example, if a container goes down, another container needs to start. Wouldn't it be easier if this behavior was handled by a system?
That's how Kubernetes comes to the rescue! Kubernetes provides you with a framework to run distributed systems resiliently. It takes care of scaling and failover for your application, provides deployment patterns, and more. For example, Kubernetes can easily manage a canary deployment for your system.
Kubernetes provides you with:

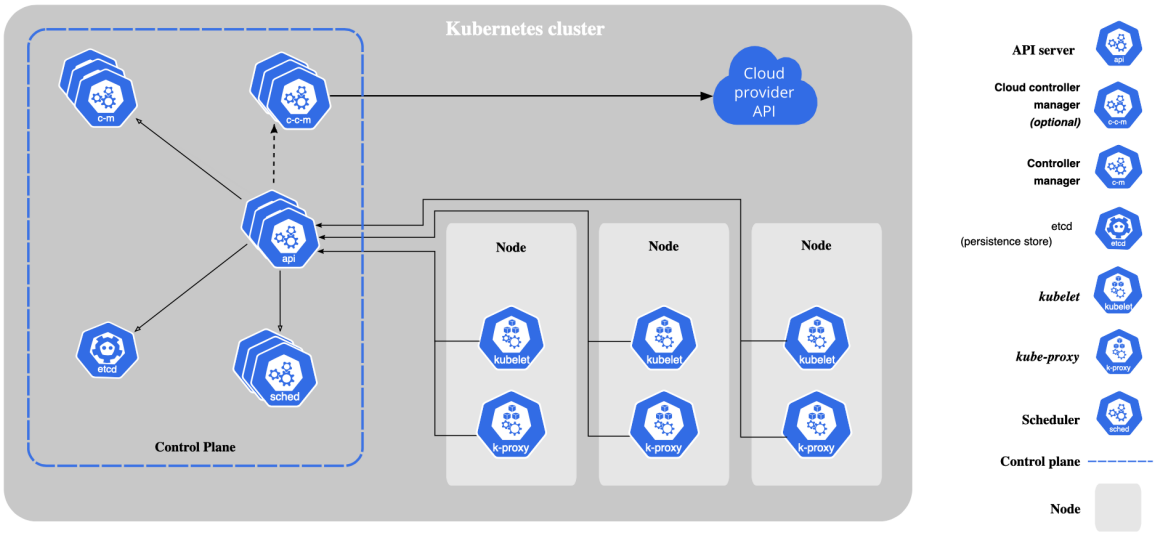
- Service discovery and load balancing Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes

is able to load balance and distribute the network traffic so that the deployment is stable.

- Storage orchestration Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.
- Automated rollouts and rollbacks You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.
- Automatic bin packing You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.
- Self-healing Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.
- Secret and configuration management Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.

Kubernetes Components

When you deploy Kubernetes, you get a cluster.
A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node.
The worker node(s) host the Pods that are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.
This document outlines the various components you need to have a complete and working Kubernetes cluster.
Here's the diagram of a Kubernetes cluster with all the components tied together.



Control Plane Components

The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new **pod** when a deployment's replicas field is unsatisfied).
Control plane components can be run on any machine in the cluster. However, for simplicity, set up scripts typically start all control plane components on the same machine, and do not run user containers on this machine.

kube-apiserver

The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane.
The main implementation of a Kubernetes API server is kube-apiserver. **kube-apiserver** is designed to scale horizontally—that is, it scales by deploying more instances. You can run several instances of kube-apiserver and balance traffic between those instances.

etcd

Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.
If your Kubernetes cluster uses etcd as its backing store, make sure you have a back up plan for those data.

kube-scheduler

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

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

kube-controller-manager

Control Plane component that runs controller processes.
Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

These controllers include:

- Node controller: Responsible for noticing and responding when nodes go down.
- Replication controller: Responsible for maintaining the correct number of pods for every replication controller object in the system.
- Endpoints controller: Populates the Endpoints object (that is, joins Services & Pods).
- Service Account & Token controllers: Create default accounts and API access tokens for new namespaces.

cloud-controller-manager

A Kubernetes control plane component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that just interact with your cluster.
The cloud-controller-manager only runs controllers that are specific to your cloud provider. If you are running Kubernetes on your own premises, or in a learning environment inside your own PC, the cluster does not have a cloud controller manager.
As with the kube-controller-manager, the cloud-controller-manager combines several logically independent control loops into a single binary that you run as a single process. You can scale horizontally (run more than one copy) to improve performance or to help tolerate failures.

The following controllers can have cloud provider dependencies:

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

Node Components

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

kubelet

An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod.
The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

kube-proxy

kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept.
kube-proxy maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.
kube-proxy uses the operating system packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself.

Container runtime

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

Addons

Addons use Kubernetes resources (DaemonSet, Deployment, etc) to implement cluster features. Because these are providing cluster-level features, namespaced resources for addons belong within the kube-system namespace.

DNS

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

Web UI (Dashboard)

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

Container Resource Monitoring

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

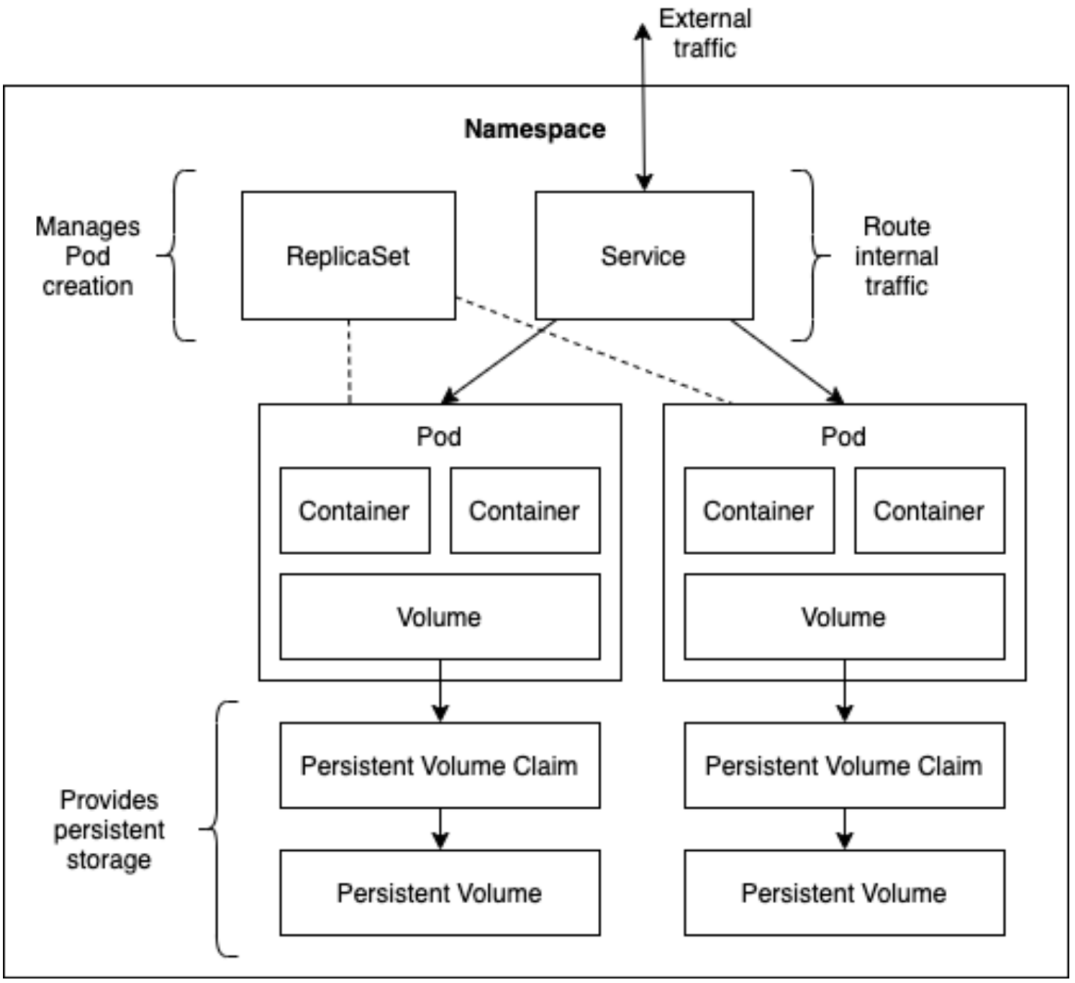
Cluster-level Logging

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

Kubernetes Architecture

Kubernetes is a very large and powerful system with many different resources and operations that can be performed on those objects. Kubernetes provides a layer of abstraction over the infrastructure and introduces the following set of basic objects which represent the desired cluster state:

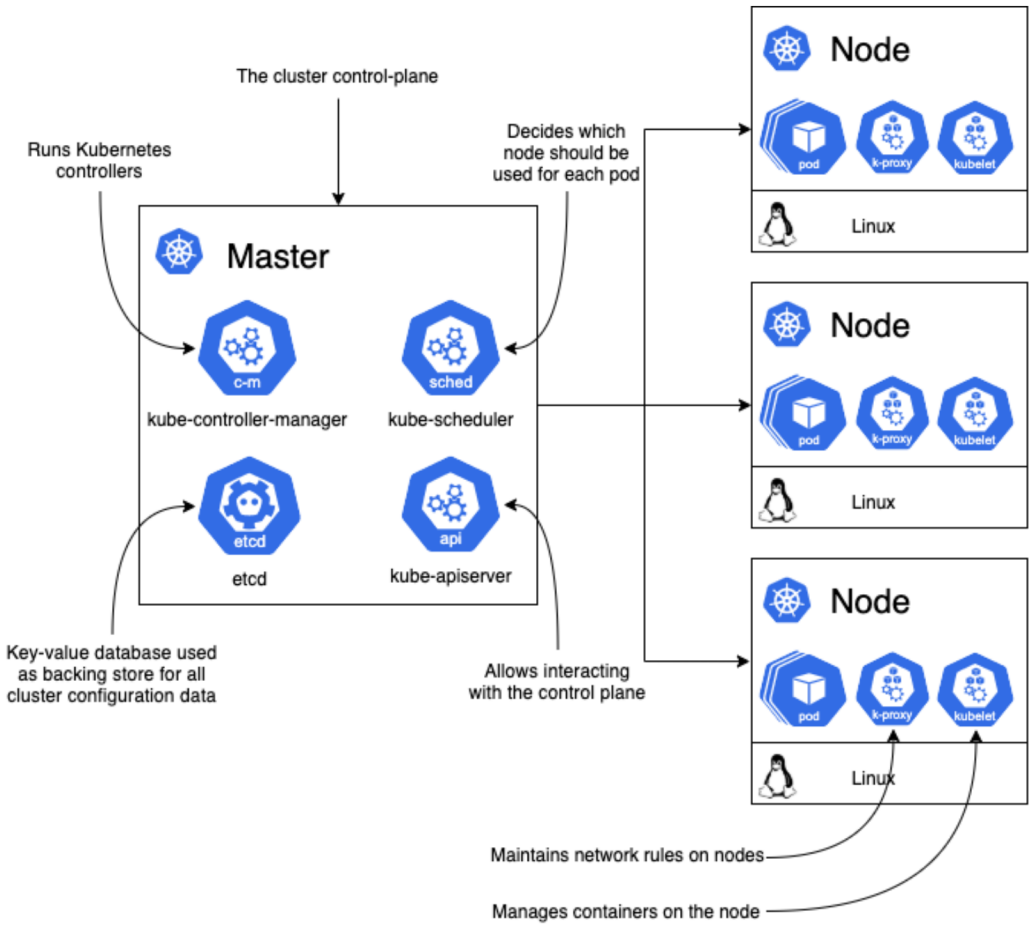
- Pod - a group of containers that are deployed together on the same host. The Pod is the smallest deployable unit on a Node and provides a way to mount storage, set environment variables, and provide other configuration information for the container. When all the Containers of a Pod exit, the Pod dies also.
- Service - an abstraction that defines a logical set of Pods and a policy by which to access them.
- Volume - represents a directory accessible to containers running in a Pod.



Kubernetes architecture is such that basic resources are used as a foundational layer for a set of higher-level resources. The higher-level resources implement features that are needed for real production use-cases that leverage/extend the functionality provided by the basic resources. In Figure 2.1, you see that the ReplicaSet resource controls the creation of one or more Pod resources. Some other examples of high-level resources include:

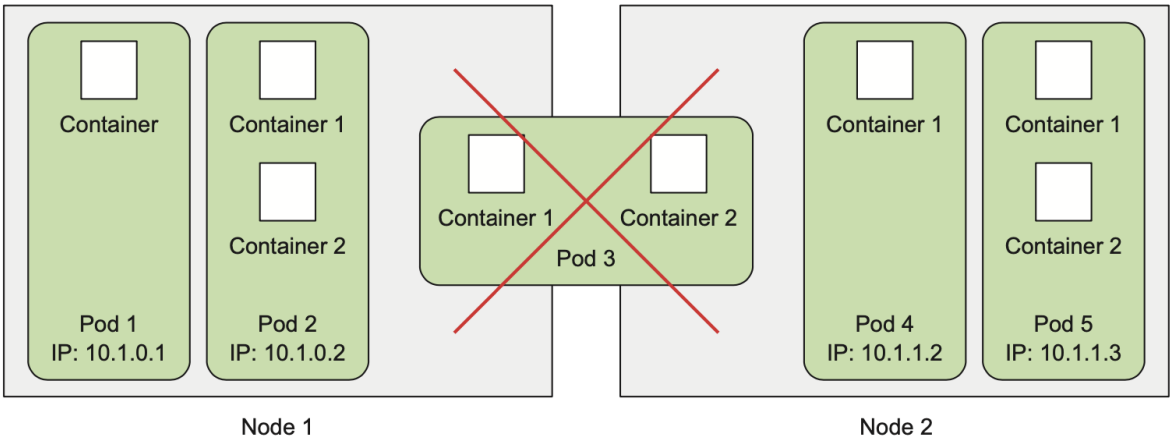
- ReplicaSet - defines that a desired number of identically configured Pods are running. If a Pod in the ReplicaSet terminates, a new Pod will be started to bring the number of running Pods back to the desired number.
- Deployments - enables declarative updates for Pods and ReplicaSets.
- Jobs - creates one or more pods that run to completion.
- CronJobs - creates Jobs on a time-based schedule

Another important Kubernetes resource are Namespaces. Most kinds of Kubernetes resources belong to one (and only one) Namespace. A namespace defines a naming scope where resources within a particular namespace must be uniquely named. Namespaces also provide a way to isolate users and applications from each other through role-based access controls (RBAC), network policies, and resource quotas. These controls allow creating a multi-tenant Kubernetes cluster where multiple users share the same cluster and avoid impacting each other (e.g., the “noisy neighbor” problem).



Introducing pods

You've already learned that a pod is a co-located group of containers and represents the basic building block in Kubernetes. Instead of deploying containers individually, you always deploy and operate on a pod of containers. We're not implying that a pod always includes more than one container—it's common for pods to contain only a single container The key thing about pods is that when a pod does contain multiple containers, all of them are always run on a single worker node—it never spans multiple worker nodes, as shown in figure



Understanding why we need pods

But why do we even need pods? Why can't we use containers directly? Why would we even need to run multiple containers together? Can't we put all our processes into a single container? We'll answer those questions now.

UNDERSTANDING WHY MULTIPLE CONTAINERS ARE BETTER THAN ONE CONTAINER RUNNING MULTIPLE PROCESSES

Imagine an app consisting of multiple processes that either communicate through *IPC* (Inter-Process Communication) or through locally stored files, which requires them to run on the same machine. Because in Kubernetes you always run processes in containers and each container is much like an isolated machine, you may think it makes sense to run multiple processes in a single container, but you shouldn't do that.
Containers are designed to run only a single process per container (unless the process itself spawns child processes). If you run multiple unrelated processes in a single container, it is your responsibility to keep all those processes running, manage their logs, and so on. For example, you'd have to include a mechanism for automatically restarting individual processes if they crash. Also, all those processes would log to the same standard output, so you'd have a hard time figuring out what process logged what. Therefore, you need to run each process in its own container. That's how Docker and Kubernetes are meant to be used.

Understanding pods

Because you're not supposed to group multiple processes into a single container, it's obvious you need another higher-level construct that will allow you to bind containers together and manage them as a single unit. This is the reasoning behind pods.

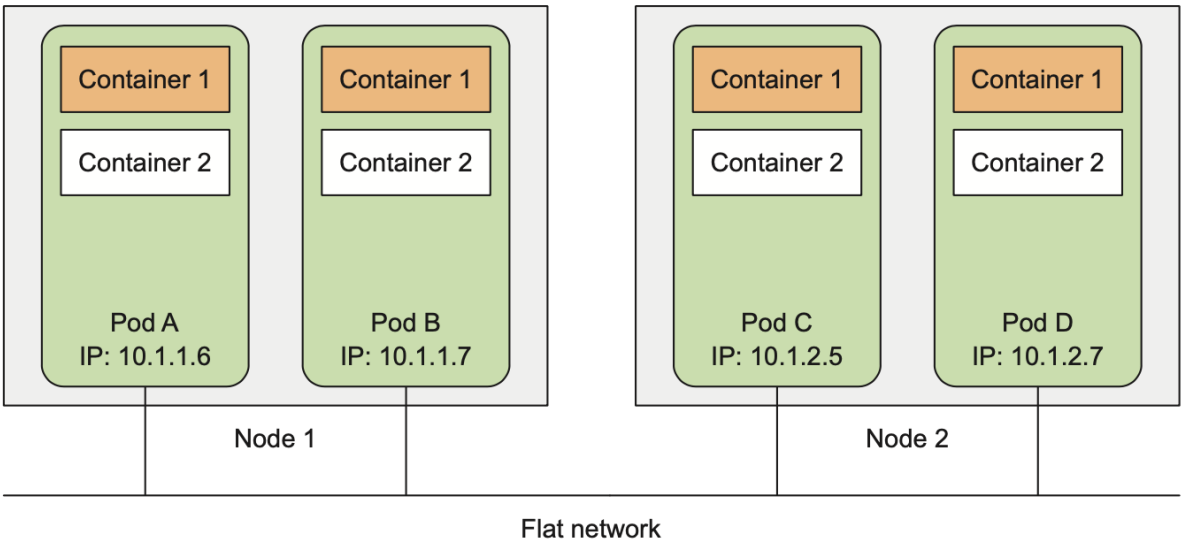
A pod of containers allows you to run closely related processes together and provide them with (almost) the same environment as if they were all running in a single container, while keeping them somewhat isolated. This way, you get the best of both worlds. You can take advantage of all the features containers provide, while at the same time giving the processes the illusion of running together.

UNDERSTANDING THE PARTIAL ISOLATION BETWEEN CONTAINERS OF THE SAME POD

We learned that containers are completely isolated from each other, but now you see that you want to isolate groups of containers instead of individual ones. You want containers inside each group to share certain resources, although not all, so that they're not fully isolated. Kubernetes achieves this by configuring Docker to have all containers of a pod share the same set of Linux namespaces instead of each container having its own set. Because all containers of a pod run under the same Network and UTS namespaces (we're talking about Linux namespaces here), they all share the same hostname and network interfaces. Similarly, all containers of a pod run under the same IPC namespace and can communicate through IPC. In the latest Kubernetes and Docker versions, they can also share the same PID namespace, but that feature isn't enabled by default.
But when it comes to the filesystem, things are a little different. Because most of the container's filesystem comes from the container image, by default, the filesystem of each container is fully isolated from other containers. However, it's possible to have them share file directories using a Kubernetes concept called a *Volume*

INTRODUCING THE FLAT INTER-POD NETWORK

All pods in a Kubernetes cluster reside in a single flat, shared, network-address space (shown in figure), which means every pod can access every other pod at the other pod's IP address. No *NAT* (Network Address Translation) gateways exist between them. When two pods send network packets between each other, they'll each see the actual IP address of the other as the source IP in the packet.



Consequently, communication between pods is always simple. It doesn't matter if two pods are scheduled onto a single or onto different worker nodes; in both cases the containers inside those pods can communicate with each other across the flat NATless network

UNDERSTANDING WHEN TO USE MULTIPLE CONTAINERS IN A POD

The main reason to put multiple containers into a single pod is when the application consists of one main process and one or more complementary processes. For example, the main container in a pod could be a web server that serves files from a certain file directory, while an additional container (a sidecar container) periodically downloads content from an external source and stores it in the web server's directory use a Kubernetes Volume that you mount into both containers.

Creating pods from YAML or JSON descriptors

Pods and other Kubernetes resources are usually created by posting a JSON or YAML manifest to the Kubernetes REST API endpoint. Also, you can use other, simpler ways of creating resources, such as the **kubectl run** command you used in the previous chapter, but they usually only allow you to configure a limited set of properties, not all. Additionally, defining all your Kubernetes objects from YAML files makes it possible to store them in a version control system, with all the benefits it brings.
To configure all aspects of each type of resource, you'll need to know and understand the Kubernetes API object definitions. You'll get to know most of them as you learn about each resource type throughout this book. We won't explain every single property, so you should

also refer to the Kubernetes API reference documentation at http://kubernetes.io/docs/reference/ when creating objects.

Examining a YAML descriptor of an existing pod

You already have some existing pods, so let's look at what a YAML definition for one of those pods looks like. You'll use the **kubectl get** command with the **-o yaml** option to get the whole YAML definition of the pod, as shown in the following listing.

```
$ kubectl get pod podname -o yaml
apiVersion: v1                                                    #A
kind: Pod                                                         #B
metadata:                                                         #C
  annotations:
    kubernetes.io/created-by: ...
  creationTimestamp: 2016-03-18T12:37:50Z
  generateName: kubia-
  labels:
    run: kubia
  name: kubia-zxzij
  namespace: default
  resourceVersion: "294"
  selfLink: /api/v1/namespaces/default/pods/kubia-zxzij
  uid: 3a564dc0-ed06-11e5-ba3b-42010af00004
spec:                                                             #D
  containers:
  - image: luksa/kubia
    imagePullPolicy: IfNotPresent
    name: kubia
    ports:
    - containerPort: 8080
      protocol: TCP
    resources:
      requests:
        cpu: 100m
   terminationMessagePath: /dev/termination-log
    volumeMounts:
    - mountPath: /var/run/secrets/k8s.io/servacc
      name: default-token-kvcqa
      readOnly: true
  dnsPolicy: ClusterFirst
  nodeName: gke-kubia-e8fe08b8-node-txje
  restartPolicy: Always
  serviceAccount: default
  serviceAccountName: default
  terminationGracePeriodSeconds: 30
  volumes:
  - name: default-token-kvcqa
    secret:
      secretName: default-token-kvcqa
status:                                                           #E
  conditions:
  - lastProbeTime: null
```

```
    lastTransitionTime: null
    status: "True"
    type: Ready
  containerStatuses:
  - containerID: docker://f0276994322d247ba...
    image: luksa/kubia
    imageID: docker://4c325bcc6b40c110226b89fe...
    lastState: {}
    name: kubia
    ready: true
    restartCount: 0
    state:
      running:
        startedAt: 2016-03-18T12:46:05Z
  hostIP: 10.132.0.4
  phase: Running
  podIP: 10.0.2.3
  startTime: 2016-03-18T12:44:32Z
```

#A Kubernetes API version used in this YAML descriptor
#B Type of Kubernetes object/resource
#C Pod metadata (name, labels, annotations, and so on)
#D Pod specification contents (list of pod's containers, volumes, and so on)
#E Detailed status of the pod and its containers

INTRODUCING THE MAIN PARTS OF A POD DEFINITION

The pod definition consists of a few parts. First, there's the Kubernetes API version used in the YAML and the type of resource the YAML is describing. Then, three important sections are found in almost all Kubernetes resources:

- ***Metadata*** includes the name, namespace, labels, and other information about the pod.
- ***Spec*** contains the actual description of the pod's contents, such as the pod's containers, volumes, and other data.
- ***Status*** contains the current information about the running pod, such as what condition the pod is in, the description and status of each container, and the pod's internal IP and other basic info.

Creating a simple YAML descriptor for a pod nxinig-pod.yaml

```
kind: Pod
apiVersion: v1
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.11
    ports:
    - containerPort: 80
      protocol: TCP
```

Using kubectl create to create the pod

To create the pod from your YAML file, use the kubectl create command:

```
$ kubectl create -f nginix-pod.yaml
pod/nginx created
```

The kubectl create -f command is used for creating any resource (not only pods) from a YAML or JSON file.

RETRIEVING THE WHOLE DEFINITION OF A RUNNING POD

After creating the pod, you can ask Kubernetes for the full YAML of the pod. You'll see it's similar to the YAML you saw earlier. You'll learn about the additional fields appearing in the returned definition in the next sections. Go ahead and use the following command to see the full descriptor of the pod:

```
$ kubectl get po nginx -o yaml
```

If you're more into JSON, you can also tell kubectl to return JSON instead of YAML like this (this works even if you used YAML to create the pod):

```
$ kubectl get po nginx -o json
```

SEEING YOUR NEWLY CREATED POD IN THE LIST OF PODS

Your pod has been created, but how do you know if it's running? Let's list pods to see their statuses:

```
$ kubectl get po
NAME                    READY   STATUS    RESTARTS   AGE
nginx                   1/1     Running   0          2m14s
```

There's your nginix pod. Its status shows that it's running. You'll probably want to confirm that's true by talking to the pod. First, you'll look at the app's log to check for any errors.

Viewing application logs

RETRIEVING A POD'S LOG WITH KUBECTL LOGS

To see your pod's log (more precisely, the container's log) you run the following command on your local machine (no need to ssh anywhere):

```
$ kubectl logs po/nginx
```

You haven't sent any web requests to your nginix app, so the log only shows a single log statement about the server starting up. As you can see, retrieving logs of an application running in Kubernetes is incredibly simple if the pod only contains a single container.

Container logs are automatically rotated daily and every time the log file reaches 10MB in size. The kubectl logs command only shows the log entries from the last rotation.

SPECIFYING THE CONTAINER NAME WHEN GETTING LOGS OF A MULTI-CONTAINER POD

If your pod includes multiple containers, you have to explicitly specify the container name by including the -c <container name> option when running kubectl logs. In your nginix pod, you set the container's name to nginix, so if additional containers exist in the pod, you'd have to get its logs like this:

```
$ kubectl logs podName -c containerName
```

Note that you can only retrieve container logs of pods that are still in existence. When a pod is deleted, its logs are also deleted. To make a pod's logs available even after the pod is deleted, you need to set up centralized, cluster-wide logging, which stores all the logs into a central store.

Sending requests to the pod

The pod is now running—at least that's what kubectl get and your app's log say. But how do you see it in action? There are some ways of connecting to a pod for testing and debugging purposes. One of them is through *port forwarding*.

FORWARDING A LOCAL NETWORK PORT TO A PORT IN THE POD

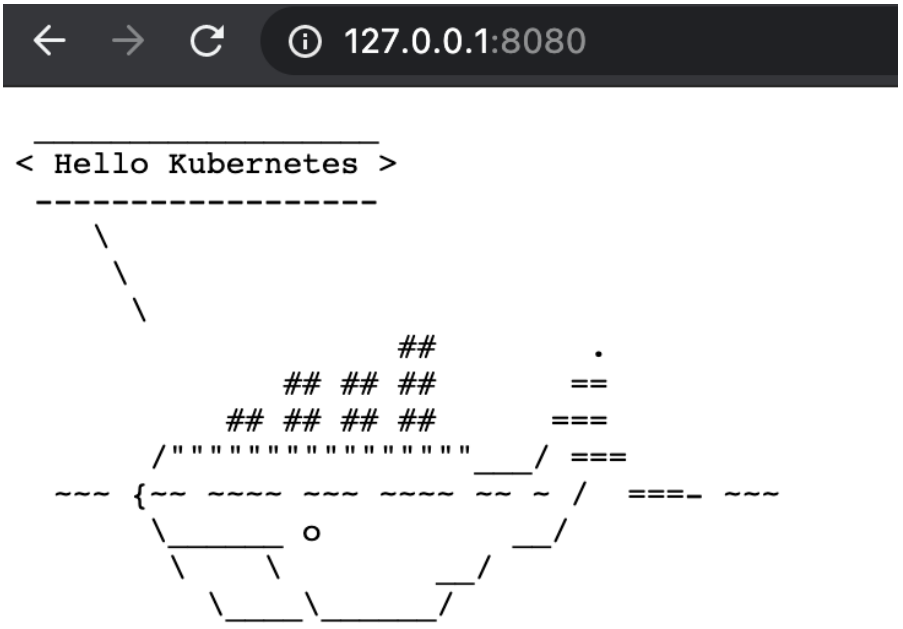
When you want to talk to a specific pod without going through a service (for debugging or other reasons), Kubernetes allows you to configure port forwarding to the pod. This is done through the kubectl port-forward command. The following command will forward your machine's local port 8080 to port 80 of your nginix pod:

```
$ kubectl get po
NAME                      READY   STATUS    RESTARTS   AGE
nginx                     1/1     Running   0          171m
$ kubectl port-forward nginx 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

Open a new shell to get the log

```
$ kubectl logs po/nginx -f
```

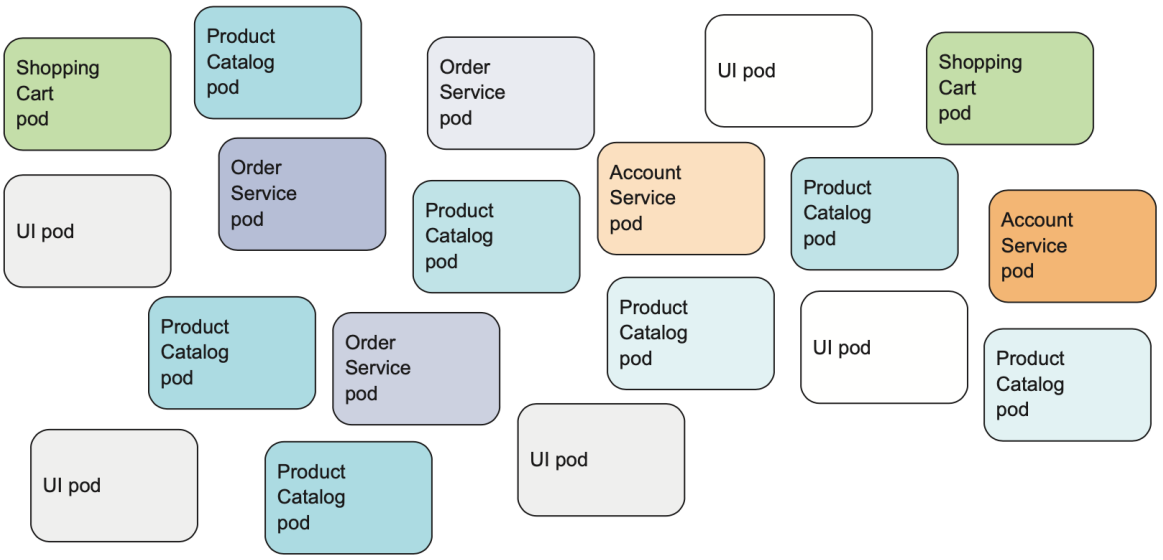
The port forwarder is running and you can now connect to your pod through the local port.



See the log:

```
$ kubectl logs po/nginx -f
127.0.0.1 - - [29/Nov/2020:21:46:19 +0000] "GET / HTTP/1.1" 200
425 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 11_0_0)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/86.0.4240.198
Safari/537.36" "-"
127.0.0.1 - -
[...]
```

At this point, you have pods running in your cluster. When deploying actual applications, most users will end up running many more pods. As the number of pods increases, the need for categorizing them into subsets becomes more and more evident.
For example, with microservices architectures, the number of deployed microser-vices can easily exceed 20 or more. Those components will probably be replicated (multiple copies of the same component will be deployed) and multiple versions or releases (stable, beta, canary, and so on) will run concurrently. This can lead to hundreds of pods in the system. Without a mechanism for organizing them, you end up with a big, incomprehensible mess, such as the one shown in figure. The figure shows pods of multiple microservices, with several running multiple replicas, and others running different releases of the same microservice.

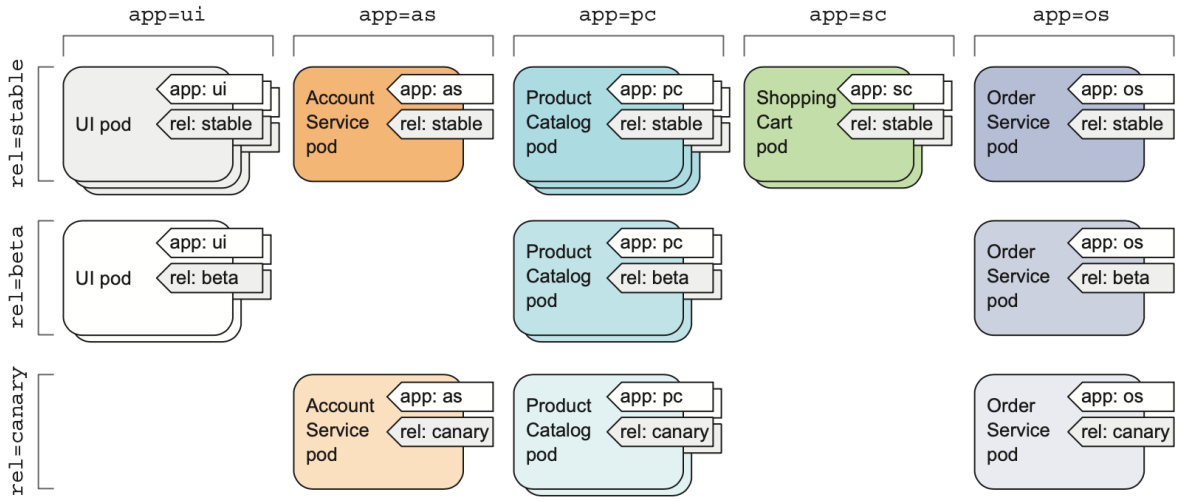


It's evident you need a way of organizing them into smaller groups based on arbitrary criteria, so every developer and system administrator dealing with your system can easily see which pod is which. And you'll want to operate on every pod belonging to a certain group with a single action instead of having to perform the action for each pod individually. Organizing pods and all other Kubernetes objects is done through *labels*.

Introducing labels

Labels are a simple, yet incredibly powerful, Kubernetes feature for organizing not only pods, but all other Kubernetes resources. A label is an arbitrary key-value pair you attach to a resource, which is then utilized when selecting resources using *label selectors* (resources are filtered based on whether they include the label specified in the selector). A resource can have more than one label, as long as the keys of those labels are unique within that resource. You usually attach labels to resources when you create them, but you can also add additional labels or even modify the values of existing labels later without having to recreate the resource.

Let's turn back to the microservices example from figure. By adding labels to those pods, you get a much-better-organized system that everyone can easily make sense of. Each pod is labeled with two labels:
app, which specifies which app, component, or microservice the pod belongs to.
rel, which shows whether the application running in the pod is a stable, beta, or a canary release.
By adding these two labels, you've essentially organized your pods into two dimensions (horizontally by app and vertically by release), as shown in figure



Specifying labels when creating a pod

Now, you'll see labels in action by creating a new pod with two labels. Create a new file called nginix-manual-with-labels.yaml with the contents of the following listing.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginix-manual-v2
  labels:
    creation_method: manual
    env: prod
spec:
  containers:
  - name: nginx
    image: nginx:1.11
```

You've included the labels creation_method=manual and env=prod in the metadata .labels section. You'll create this pod now:

```
$ kubectl create -f nginix-manual-with-labels.yaml
pod/nginix-manual-v2 created
$ kubectl get po
NAME                READY    STATUS     RESTARTS    AGE
```

```
nginix-manual-v2   1/1     Running   0          5s
```

The kubectl get pods command doesn't list any labels by default, but you can see them by using the --show-labels switch:

```
$ kubectl get po --show-labels
NAME               READY   STATUS    RESTARTS   AGE    LABELS
nginix-manual-v2   1/1     Running   0          101s
creation_method=manual,env=prod
```

Modifying labels of existing pods

Labels can also be added to and modified on existing pods. Because the kubia-man- ual pod was also created manually, let's add the creation_method=manual label to it:

```
$ kubectl get po --show-labels
NAME               READY   STATUS    RESTARTS   AGE    LABELS
nginix-manual-v2   1/1     Running   0          4m3s
creation_method=manual,env=prod
nginx              1/1     Running   0          12s    <none>
```

Now, let's also change the env=prod label to env=debug on the nginix-manual-v2 pod, to see how existing labels can be changed.

```
$ kubectl label po nginix-manual-v2 env=debug --overwrite
pod/nginix-manual-v2 labeled
$ kubectl get po --show-labels
NAME               READY   STATUS    RESTARTS   AGE     LABELS
nginix-manual-v2   1/1     Running   0          5m13s
creation_method=manual,env=debug
```

Now, let's add a new label to the nginix pod create previously:

```
$ kubectl label po nginx env=debug --overwrite
pod/nginx labeled
$ kubectl get po --show-labels
NAME               READY   STATUS    RESTARTS   AGE     LABELS
nginix-manual-v2   1/1     Running   0          5m13s
creation_method=manual,env=prod
nginx              1/1     Running   0          82s     env=debug
```

As you can see, attaching labels to resources is trivial, and so is changing them on existing resources. It may not be evident right now, but this is an incredibly powerful feature. But first, let's see what you can do with these labels, in addition to displaying them when listing pods.

Listing pods using a label selector

Let's use label selectors on the pods you've created so far. To see all pods you created manually (you labeled them with creation_method=manual), do the following:

```
$ kubectl get po -l creation_method=manual
NAME                READY   STATUS    RESTARTS   AGE
nginix-manual-v2    1/1     Running   0          7m29s
```

To list all pods that include the env label, whatever its value is:

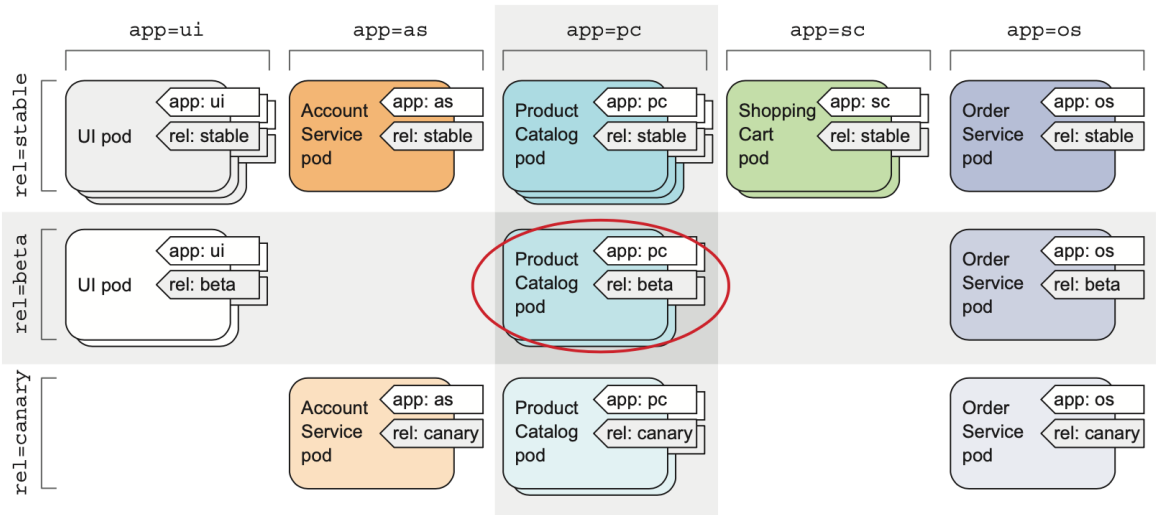
```
$ kubectl get po -l env
NAME                READY   STATUS    RESTARTS   AGE
nginix-manual-v2    1/1     Running   0          8m37s
nginx               1/1     Running   0          4m46s
```

And those that don't have the env label:

```
$ kubectl get po -l '!env'
No resources found in default namespace.
```

Using multiple conditions in a label selector

A selector can also include multiple comma-separated criteria. Resources need to match all of them to match the selector. If, for example, you want to select only pods running the beta release of the product catalog microservice, you'd use the following selector:
app=pc,rel=beta
Label selectors aren't useful only for listing pods, but also for performing actions on a subset of all pods. For example, we can use label selectors to delete multiple pods at once. But label selectors aren't used only by kubectl. They're also used internally.



Using labels and selectors to constrain pod scheduling

All the pods you've created so far have been scheduled pretty much randomly across your worker nodes. As I've mentioned in the previous chapter, this is the proper way of working in a Kubernetes cluster. Because Kubernetes exposes all the nodes in the cluster as a single, large deployment platform, it shouldn't matter to you what node a pod is scheduled to. Because each pod gets the exact amount of computational resources it requests (CPU, memory, and so on) and its accessibility from other pods isn't at all affected by the node the pod is scheduled to, usually there shouldn't be any need for you to tell Kubernetes exactly where to schedule your pods.
Certain cases exist, however, where you'll want to have at least a little say in where a pod should be scheduled. A good example is when your hardware infrastructure isn't homogenous. If part of your worker nodes have spinning hard drives, whereas others have SSDs, you may want to schedule certain pods to one group of nodes and the rest to the other. Another example is when you need to schedule pods performing intensive GPU-based computation only to nodes that provide the required GPU acceleration.
You never want to say specifically what node a pod should be scheduled to, because that would couple the application to the infrastructure, whereas the whole idea of Kubernetes is hiding the actual infrastructure from the apps that run on it. But if you want to have a say in where a pod should be scheduled, instead of specifying an exact node, you should describe the node requirements and then let Kubernetes select a node that matches those requirements. This can be done through node labels and node label selectors.

Using labels for categorizing worker nodes

As you learned earlier, pods aren't the only Kubernetes resource type that you can attach a label to. Labels can be attached to any Kubernetes object, including nodes. Usually, when the ops team adds a new node to the cluster, they'll categorize the node by attaching labels specifying the type of hardware the node provides or anything else that may come in handy when scheduling pods.
Let's imagine one of the nodes in your cluster contains a GPU meant to be used for general-purpose GPU computing. You want to add a label to the node showing this feature. You're going to add the label gpu=true to one of your nodes (pick one out of the list returned by kubectl get nodes):

```
$ kubectl label node gke-node01-85f6-node-0rrx gpu=true
node "gke-node01-85f6-node-0rrx" labeled
```

Now you can use a label selector when listing the nodes, like you did before with pods. List only nodes that include the label gpu=true:

```
$ kubectl get nodes -l gpu=true
NAME                        STATUS AGE
gke-node01-85f6-node-0rrx  Ready  1d
```

Scheduling pods to specific nodes

Now imagine you want to deploy a new pod that needs a GPU to perform its work. To ask the scheduler to only choose among the nodes that provide a GPU, you'll add a node selector to the pod's YAML.

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: nginix-gpu
spec:
  nodeSelector:                #A
    gpu: "true"
  containers:
  - image: nginx:1.11
    name: nginix
```

#A nodeSelector tells Kubernetes to deploy this pod only to nodes containing the gpu=true label.

You've added a nodeSelector field under the spec section. When you create the pod, the scheduler will only choose among the nodes that contain the gpu=true label

Using namespaces to group resources

Kubernetes groups objects into **namespaces**. These aren't the Linux namespaces, which are used to isolate processes from each other. Kubernetes namespaces provide a scope for objects names. Instead of having all your resources in one single namespace, you can split them into multiple namespaces, which also allows you to use the same resource names multiple times (across different namespaces).
Using multiple namespaces allows you to split complex systems with numerous components into smaller distinct groups. They can also be used for separating resources in a multi-tenant environment, splitting up resources into production, development, and QA environments, or in any other way you may need. Resource names only need to be unique within a namespace. Two different namespaces can contain resources of the same name. But, while most types of resources are namespaced, a few aren't. One of them is the Node resource, which is global and not tied to a single namespace.

Let's see how to use namespaces now.

First, let's list all namespaces in your cluster:

```
$ kubectl get ns
NAME              STATUS   AGE
default           Active   57d
kube-node-lease   Active   57d
kube-public       Active   57d
kube-system       Active   57d
```

Up to this point, you've operated only in the default namespace. When listing resources with the kubectl get command, you've never specified the namespace explicitly, so kubectl always defaulted to the default namespace, showing you only the objects in that namespace. But as you can see from the list, the kube-public and the kube-system namespaces also exist. Let's look at the pods that belong to the kube-system name- space, by telling kubectl to list pods in that namespace only:

```
$ kubectl get po --namespace kube-system
NAME                               READY   STATUS    RESTARTS   AGE
coredns-f9fd979d6-5thdd            1/1     Running   3          57d
etcd-minikube                      1/1     Running   3          57d
kube-apiserver-minikube            1/1     Running   3          57d
kube-controller-manager-minikube   1/1     Running   3          57d
kube-proxy-grxkg                   1/1     Running   3          57d
kube-scheduler-minikube            1/1     Running   3          57d
storage-provisioner                1/1     Running   7          57d
```

Namespaces enable you to separate resources that don't belong together into non overlapping groups. If several users or groups of users are using the same Kubernetes cluster, and they each manage their own distinct set of resources, they should each use their own namespace. This way, they don't need to take any special care not to inadvertently modify or delete the other users' resources and don't need to concern themselves with name conflicts, because namespaces provide a scope for resource names, as has already been mentioned.
Besides isolating resources, namespaces are also used for allowing only certain users access to particular resources and even for limiting the amount of computational resources available to individual users.

Creating a namespace

A namespace is a Kubernetes resource like any other, so you can create it by posting a YAML file to the Kubernetes API server. Let's see how to do this now.

CREATING A NAMESPACE FROM A YAML FILE

First, create a custom-namespace.yaml file with the following listing's contents (you'll find the file in the book's code archive).

```
apiVersion: v1
kind: Namespace                #A
metadata:
  name: custom-namespace       #B
```

#A This says you're defining a namespace.
#B This is the name of the namespace.

CREATING A NAMESPACE WITH KUBECTL CREATE NAMESPACE

Although writing a file like the previous one isn't a big deal, it's still a hassle. Luckily, you can also create namespaces with the dedicated kubectl create namespace command, which is quicker than writing a YAML file. By having you create a YAML manifest for the namespace, I wanted to reinforce the idea that everything in Kubernetes has a corresponding API object that you can create, read, update, and delete by posting a YAML manifest to the API server. You could have created the namespace like this:

```
 $ kubectl create namespace custom-namespace
        namespace "custom-namespace" created
```

Managing objects in other namespaces

To create resources in the namespace you've created, either add a namespace: custom-namespace entry to the metadata section, or specify the namespace when creating the resource with the kubectl create command:

```
$ kubectl create -f nginix-pod.yaml -n custom-namespace
  pod "nginix" created
```

You now have two pods with the same name (nginix). One is in the default namespace, and the other is in your custom-namespace.
When listing, describing, modifying, or deleting objects in other namespaces, you need to pass the **--namespace** (or **-n**) flag to **kubectl**. If you don't specify the name- space, kubectl performs the action in the default namespace configured in the current kubectl context.
To quickly switch to a different namespace, you can set up the following alias: `alias kcd='kubectl config set-context $(kubectl config current-context) --namespace '`. You can then switch between namespaces using kcd some-namespace.

Understanding the isolation provided by namespaces

To wrap up this section about namespaces, let me explain what namespaces don't provide—at least not out of the box. Although namespaces allow you to isolate objects into distinct groups, which allows you to operate only on those belonging to the specified namespace, they don't provide any kind of isolation of running objects.
For example, you may think that when different users deploy pods across different namespaces, those pods are isolated from each other and can't communicate, but that's not necessarily the case. Whether namespaces provide network isolation depends on which networking solution is deployed with Kubernetes. When the solution doesn't provide inter-namespace network isolation, if a pod in namespace **foo** knows the IP address of a pod in namespace bar, there is nothing preventing it from sending traffic, such as HTTP requests, to the other pod.

Stopping and removing pods

You've created a number of pods, which should all still be running. You have pods running in the default namespace and one pod in custom-namespace. You're going to stop all of them now, because you don't need them anymore.

Deleting a pod by lable

First, delete the nginix pod by name:

```
$ kubectl delete po -l env=debug
  pod "nginx" deleted
```

Deleting a pod by name

First, delete the nginix pod by name:

```
$ kubectl delete po nginix-manual-v2
  pod "nginix-manual-v2" deleted
```

By deleting a pod, you're instructing Kubernetes to terminate all the containers that are part of that pod. Kubernetes sends a SIGTERM signal to the process and waits a certain number of seconds (30 by default) for it to shut down gracefully. If it doesn't shut down in time, the process is then killed through SIGKILL. To make sure your processes are always shut down gracefully, they need to handle the SIGTERM signal properly.

Deleting pods by deleting the whole namespace

Okay, back to your real pods. What about the pod in the custom-namespace? You no longer need either the pods in that namespace, or the namespace itself. You can delete the whole namespace (the pods will be deleted along with the namespace auto- matically), using the following command:

```
$ kubectl delete ns custom-namespace
namespace "custom-namespace" deleted
```

Replication and other controllers

As you've learned so far, pods represent the basic deployable unit in Kubernetes. You know how to create, supervise, and manage them manually. But in real-world use cases, you want your deployments to stay up and running automatically and remain healthy without any manual intervention. To do this, you almost never create pods directly. Instead, you create other types of resources, such as **ReplicationControllers, ReplicaSet** or **Deployments**, which then create and manage the actual pods.
When you create unmanaged pods (such as the ones you created in the previous chapter), a cluster node is selected to run the pod and then its containers are run on that node. In this chapter, you'll learn that Kubernetes then monitors those containers and automatically restarts them if they fail. But if the whole node fails, the pods on the node are lost and will not be replaced with new ones, unless those pods are managed by the previously mentioned ReplicationControllers or similar. In this chapter, you'll learn how Kubernetes checks if a container is still alive and restarts it if it isn't. You'll also learn how to run managed pods—both those that run indefinitely and those that perform a single task and then stop.

Keeping pods healthy

One of the main benefits of using Kubernetes is the ability to give it a list of containers and let it keep those containers running somewhere in the cluster. You do this by creating a Pod resource and letting Kubernetes pick a worker node for it and run the pod's containers on that node. But what if one of those containers dies? What if all containers of a pod die?
As soon as a pod is scheduled to a node, the Kubelet on that node will run its containers and, from then on, keep them running as long as the pod exists. If the container's main process crashes, the Kubelet will restart the container. If your application has a bug that causes it to crash every once in a while, Kubernetes will restart it automatically, so even without doing anything special in the app itself, running the app in Kubernetes automatically gives it the ability to heal itself.
But sometimes apps stop working without their process crashing. For example, a Java app with a memory leak will start throwing OutOfMemoryErrors, but the JVM process will keep running. It would be great to have a way for an app to signal to Kubernetes that it's no longer functioning properly and have Kubernetes restart it.
We've said that a container that crashes is restarted automatically, so maybe you're thinking you could catch these types of errors in the app and exit the process when they occur. You can certainly do that, but it still doesn't solve all your problems.
For example, what about those situations when your app stops responding because it falls into an infinite loop or a deadlock? To make sure applications are restarted in such cases, you must check an application's health from the outside and not depend on the app doing it internally.

Introducing liveness probes

Kubernetes can check if a container is still alive through *liveness probes*. You can specify a liveness probe for each container in the pod's specification. Kubernetes will periodically execute the probe and restart the container if the probe fails.

Kubernetes can probe a container using one of the three mechanisms:

- An *HTTP GET* probe performs an HTTP GET request on the container's IP address, a port and path you specify. If the probe receives a response, and the response code doesn't represent an error (in other words, if the HTTP response code is 2xx or 3xx), the probe is considered successful. If the server returns an error response code or if it doesn't respond at all, the probe is considered a failure and the container will be restarted as a result.
- A *TCP Socket* probe tries to open a TCP connection to the specified port of the container. If the connection is established successfully, the probe is successful. Otherwise, the container is restarted.
- An *Exec* probe executes an arbitrary command inside the container and checks the command's exit status code. If the status code is 0, the probe is successful. All other codes are considered failures.

Creating an HTTP-based liveness probe

Let's see how to add a liveness probe to a Node.js app. Because it's a web app, it makes sense to add a liveness probe that will check whether its web server is serving requests. But because this particular Node.js app is too simple to ever fail, you'll need to make the app fail artificially.

To properly demo liveness probes, you'll modify the app slightly and make it return a 500 Internal Server Error HTTP status code for each request after the fifth one— your app will handle the first five client requests properly and then return an error on every subsequent request. Thanks to the liveness probe, it should be restarted when that happens, allowing it to properly handle client requests again.

You'll create a new pod that includes an HTTP GET liveness probe. The following listing shows the YAML for the pod.

```
apiVersion: v1
kind: Pod
metadata:
  name: kubia-liveness
spec:
  containers:
  - image: luksa/kubia-unhealthy     #A
    name: kubia
    livenessProbe:                   #B
      httpGet:
        path: /                      #C
        port: 8080                   #D
```

#A **This is the image containing the (somewhat) broken app.**
#B **A liveness probe that will perform an HTTP GET**
#C **The path to request in the HTTP request**
#D **The network port the probe should connect to**

The pod descriptor defines an httpGet liveness probe, which tells Kubernetes to periodically perform HTTP GET requests on path / on port 8080 to determine if the container is still healthy. These requests start as soon as the container is run.
After five such requests (or actual client requests), your app starts returning HTTP status code 500, which Kubernetes will treat as a probe failure, and will thus restart the container.

kubia-unhealthy Dockerfile

```
FROM node:7
ADD app.js /app.js
ENTRYPOINT ["node", "app.js"]
```

app.js

```
const http = require('http');
const os = require('os');

console.log("Kubia server starting...");

var requestCount = 0;

var handler = function(request, response) {
  console.log("Received request from " +
request.connection.remoteAddress);
  requestCount++;
  if (requestCount > 5) {
    response.writeHead(500);
    response.end("I'm not well. Please restart me!");
    return;
  }
  response.writeHead(200);
  response.end("You've hit " + os.hostname() + "\n");
};

var www = http.createServer(handler);
www.listen(8080);
```

Seeing a liveness probe in action

To see what the liveness probe does, try creating the pod now. After about a minute and a half, the container will be restarted. You can see that by running kubectl get:

```
$ kubectl create -f kubia-liveness-probe.yaml
pod/kubia-liveness created

$ kubectl get po kubia-liveness
NAME             READY   STATUS    RESTARTS   AGE
kubia-liveness   1/1     Running   1          3m

$ kubectl get po kubia-liveness
NAME             READY   STATUS    RESTARTS   AGE
kubia-liveness   1/1     Running   2          4m45s
```

The RESTARTS column shows that the pod's container has been restarted once (if you wait another minute and a half, it gets restarted again, and then the cycle continues indefinitely).

Obtaining the application log of a crashed container

In the previous chapter, you learned how to print the application's log with kubectl logs. If your container is restarted, the kubectl logs command will show the log of the current container.

When you want to figure out why the previous container terminated, you'll want to see those logs instead of the current container's logs. This can be done by using the --previous option:

```
$ kubectl logs mypod --previous
```

You can see why the container had to be restarted by looking at what kubectl describe prints out, as shown in the following listing.

```
$ kubectl describe po kubia-liveness

Name:         kubia-liveness
Namespace:    default
Priority:     0
Node:         minikube/192.168.64.2
Start Time:   Thu, 03 Dec 2020 21:46:49 +0100
Labels:       <none>
Annotations:  <none>
Status:       Running      ##The container is currently running.
IP:           172.17.0.3
IPs:
  IP:  172.17.0.3
Containers:
  kubia:
    Container ID:
docker://bb980c5d9aea818da9635555923a7abebc4fa7c12c1b6ab0baecb6281
0adc5c0
    Image:          luksa/kubia-unhealthy
```

```
    Image ID:
docker-pullable://luksa/kubia-unhealthy@sha256:5c746a42612be612094
17d913030d97555cff0b8225092908c57634ad7c235f7
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Thu, 03 Dec 2020 21:51:27 +0100
    Last State:     Terminated
      Reason:       Error
      Exit Code:    137 ##The previous container terminated with an
error and exited with code 137.
      Started:      Thu, 03 Dec 2020 21:49:37 +0100
      Finished:     Thu, 03 Dec 2020 21:51:25 +0100
    Ready:          True
    Restart Count:  1   ##The container has been restarted once
    Liveness:       http-get http://:8080/ delay=0s timeout=1s
period=10s #success=1 #failure=3
[...]
```

You can see that the container is currently running, but it previously terminated because of an error. The exit code was 137, which has a special meaning—it denotes that the process was terminated by an external signal. The number 137 is a sum of two numbers: 128+x, where x is the signal number sent to the process that caused it to terminate. In the example, x equals 9, which is the number of the SIGKILL signal, meaning the process was killed forcibly.
The events listed at the bottom show why the container was killed—Kubernetes detected the container was unhealthy, so it killed and re-created it.

Configuring additional properties of the liveness probe

You may have noticed that **kubectl describe** also displays additional information about the liveness probe:

```
Liveness:       http-get http://:8080/ delay=0s timeout=1s
period=10s #success=1 #failure=3
```

Beside the liveness probe options you specified explicitly, you can also see additional properties, such as delay, timeout, period, and so on. The delay=0s part shows that the probing begins immediately after the container is started. The timeout is set to only 1 second, so the container must return a response in 1 second or the probe is counted as failed. The container is probed every 10 seconds (period=10s) and the container is restarted after the probe fails three consecutive times (#failure=3).
These additional parameters can be customized when defining the probe. For example, to set the initial delay, add the initialDelaySeconds property to the live- ness probe as shown in the following listing.

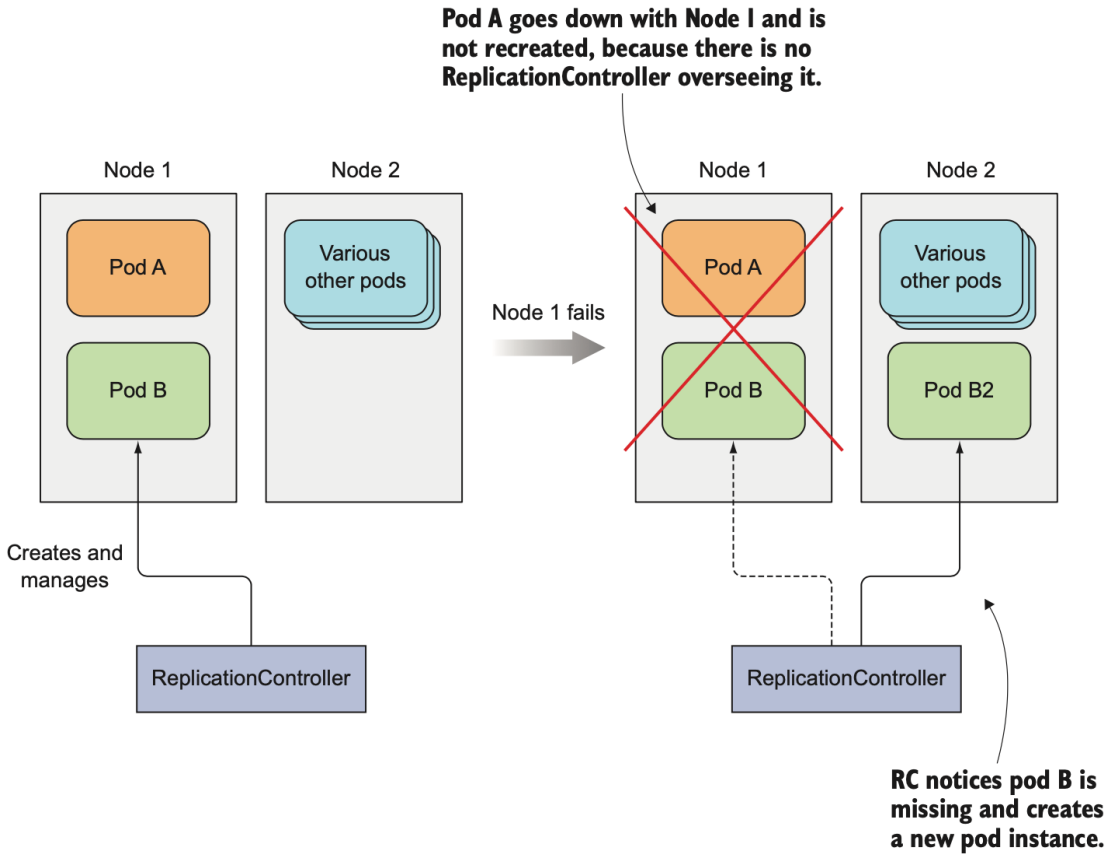
```
apiVersion: v1
kind: Pod
metadata:
  name: kubia-liveness
spec:
```

```
containers:
- image: luksa/kubia-unhealthy
  name: kubia
  livenessProbe:
    httpGet:
      path: /
      port: 8080
    initialDelaySeconds: 15
```

If you don't set the initial delay, the prober will start probing the container as soon as it starts, which usually leads to the probe failing, because the app isn't ready to start receiving requests. If the number of failures exceeds the failure threshold, the container is restarted before it's even able to start responding to requests properly.

Introducing ReplicationControllers

A ReplicationController is a Kubernetes resource that ensures its pods are always kept running. If the pod disappears for any reason, such as in the event of a node disappearing from the cluster or because the pod was evicted from the node, the ReplicationController notices the missing pod and creates a replacement pod.
Figure shows what happens when a node goes down and takes two pods with it. Pod A was created directly and is therefore an unmanaged pod, while pod B is manage



by a ReplicationController. After the node fails, the ReplicationController creates a new pod (pod B2) to replace the missing pod B, whereas pod A is lost completely— nothing will ever recreate it.
The ReplicationController in the figure manages only a single pod, but ReplicationControllers, in general, are meant to create and manage multiple copies (replicas) of a pod. That's where ReplicationControllers got their name from.

The operation of a ReplicationController

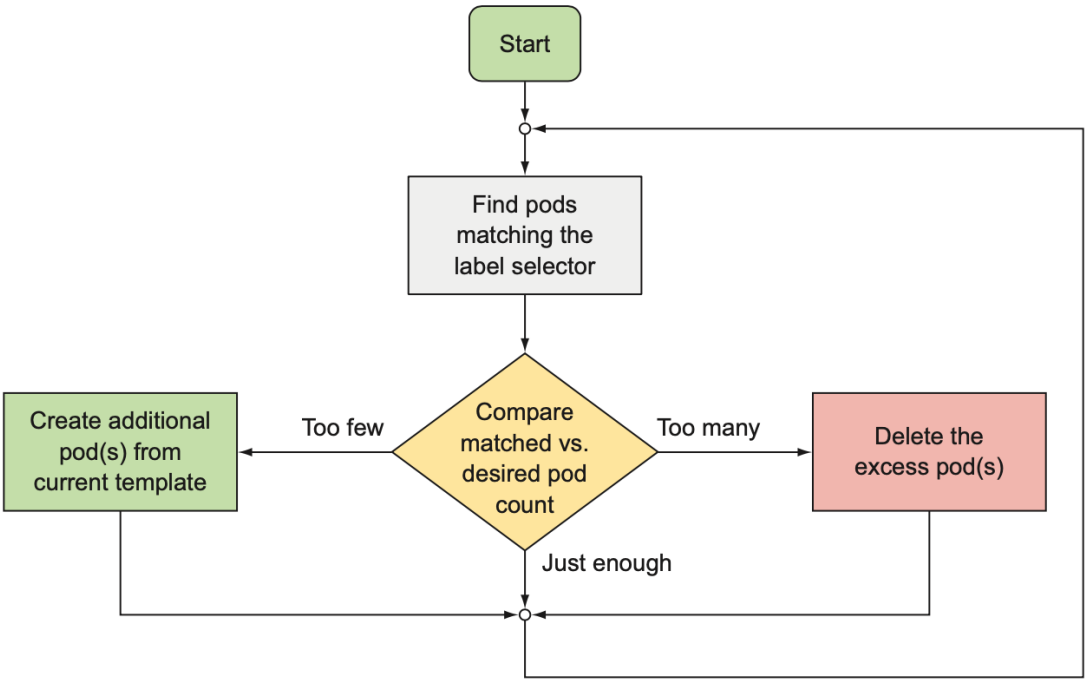
A ReplicationController constantly monitors the list of running pods and makes sure the actual number of pods of a "type" always matches the desired number. If too few such pods are running, it creates new replicas from a pod template. If too many such pods are running, it removes the excess replicas.
You might be wondering how there can be more than the desired number of replicas. This can happen for a few reasons:

- Someone creates a pod of the same type manually.
- Someone changes an existing pod's "type."
- Someone decreases the desired number of pods, and so on.

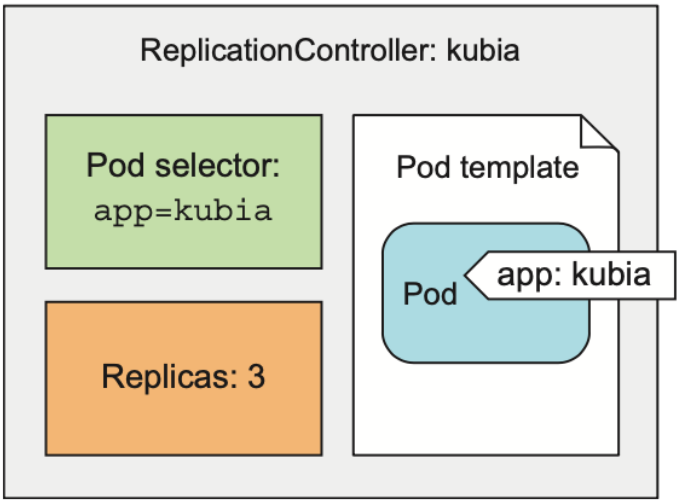
I've used the term pod "type" a few times. But no such thing exists. ReplicationControllers don't operate on pod types, but on sets of pods that match a certain label selector (you learned about them in the previous chapter).

INTRODUCING THE CONTROLLER'S RECONCILIATION LOOP

A ReplicationController's job is to make sure that an exact number of pods always matches its label selector. If it doesn't, the ReplicationController takes the appropriate action to reconcile the actual with the desired number. The operation of a ReplicationController is shown in figure



UNDERSTANDING THE THREE PARTS OF A REPLICATIONCONTROLLER
A ReplicationController has three essential parts (also shown in figure):
A *label selector*, which determines what pods are in the ReplicationController's scope A *replica count*, which specifies the desired number of pods that should be running A *pod template*, which is used when creating new pod replicas



A ReplicationController's replica count, the label selector, and even the pod template can all be modified at any time, but only changes to the replica count affect existing pods.

UNDERSTANDING THE BENEFITS OF USING A REPLICATIONCONTROLLER

Like many things in Kubernetes, a ReplicationController, although an incredibly simple concept, provides or enables the following powerful features:

- It makes sure a pod (or multiple pod replicas) is always running by starting a new pod when an existing one goes missing.
- When a cluster node fails, it creates replacement replicas for all the pods that were running on the failed node (those that were under the ReplicationController's control).
- It enables easy horizontal scaling of pods—both manual and automatic

Creating a ReplicationController

Let's look at how to create a ReplicationController and then see how it keeps your pods running. Like pods and other Kubernetes resources, you create a ReplicationController by posting a JSON or YAML descriptor to the Kubernetes API server.
You're going to create a YAML file called **kubia-rc.yaml** for your ReplicationController, as shown in the following listing.

```
apiVersion: v1
kind: ReplicationController          #A
metadata:
  name: kubia                    #B
spec:
  replicas: 3                    #C
  selector:                      #D
    app: kubia
  template:                      #E
    metadata:
      labels:
        app: kubia
    spec:
      containers:
      - name: kubia
        image: luksa/kubia
        ports:
        - containerPort: 8080
```

#A This manifest defines a ReplicationController (RC)
#B The name of this ReplicationController
#C The desired number of pod instances
#D The pod selector determining what pods the RC is operating on
#E The pod template for creating new pods

When you post the file to the API server, Kubernetes creates a new ReplicationController named kubia, which makes sure three pod instances always match the label selector app=kubia. When there aren't enough pods, new pods will be created from the provided pod template.
The pod labels in the template must obviously match the label selector of the ReplicationController; otherwise the controller would create new pods indefinitely, because spinning up a new pod wouldn't bring the actual replica count any closer to the desired number of replicas. To prevent such scenarios, the API server verifies the ReplicationController definition and will not accept it if it's misconfigured.
Not specifying the selector at all is also an option. In that case, it will be configured automatically from the labels in the pod template.

To create the ReplicationController, use the kubectl create command, which you already know:

```
$ kubectl create -f kubia-rc.yaml
replicationcontroller/kubia created
```

Seeing the ReplicationController in action

Because no pods exist with the app=kubia label, the ReplicationController should spin up three new pods from the pod template. List the pods to see if the ReplicationController has done what it's supposed to:

```
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
kubia-pp247   1/1     Running   0          18s
kubia-wjlhg   1/1     Running   0          18s
kubia-xx2v2   1/1     Running   0          18s
```

Indeed, it has! You wanted three pods, and it created three pods. It's now managing those three pods. Next you'll mess with them a little to see how the ReplicationController responds.

SEEING THE REPLICATIONCONTROLLER RESPOND TO A DELETED POD

First, you'll delete one of the pods manually to see how the ReplicationController spins up a new one immediately, bringing the number of matching pods back to three:

```
$ kubectl delete pod kubia-pp247
pod "kubia-pp247" deleted
```

Listing the pods again shows four of them, because the one you deleted is terminating, and a new pod has already been created:

```
$ kubectl get pods
NAME          READY   STATUS               RESTARTS   AGE
kubia-pp247   1/1     Terminiting          0          3m
kubia-9z8qm   0/1     ContainerCreating    0          50s
kubia-wjlhg   1/1     Running              0          3m30s
kubia-xx2v2   1/1     Running              0          3m30s
```

```
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
kubia-9z8qm   1/1     Running   0          12m
kubia-wjlhg   1/1     Running   0          15m
kubia-xx2v2   1/1     Running   0          15m
```

GETTING INFORMATION ABOUT A REPLICATIONCONTROLLER

Now, let's see what information the kubectl get command shows for ReplicationControllers:

```
$ kubectl get rc ##shorthand for replicationcontroller
NAME    DESIRED   CURRENT   READY   AGE
kubia   3         3         3       15m
```

You can see additional information about your ReplicationController with the kubectl describe command, as shown in the following listing.

```
$ kubectl describe rc kubia
Name:         kubia
Namespace:    default
Selector:     app=kubia
Labels:       app=kubia
Annotations:  <none>
Replicas:     3 current / 3 desired                                    #A

Pods Status:  3 Running / 0 Waiting / 0 Succeeded / 0 Failed    #B
Pod Template:
  Labels:  app=kubia
  Containers:
[...]
Events:                                                                #C
  Type    Reason            Age   From                    Message
  ----    ------            ----  ----                    -------
  Normal  SuccessfulCreate  19m   replication-controller  Created pod: kubia-wjlhg
  Normal  SuccessfulCreate  19m   replication-controller  Created pod: kubia-pp247
  Normal  SuccessfulCreate  19m   replication-controller  Created pod: kubia-xx2v2
  Normal  SuccessfulCreate  16m   replication-controller  Created pod: kubia-9z8qm
```

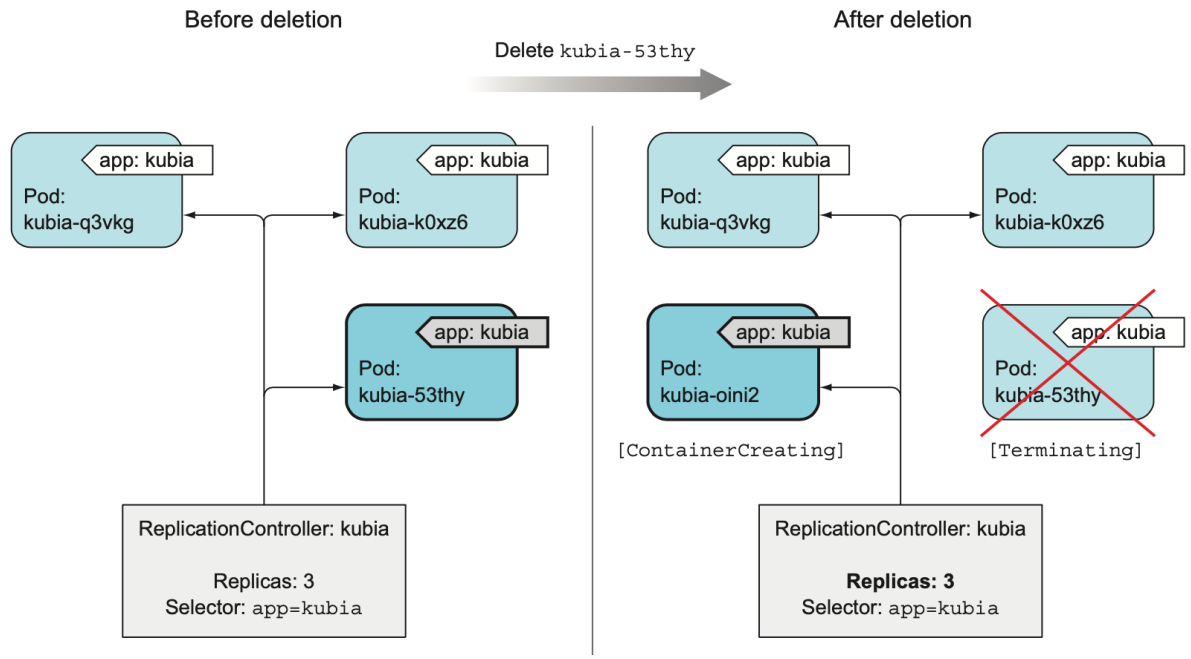
#A The actual vs. the desired number of pod instances
#B Number of pod instances per pod status
#C The events related to this ReplicationController

The current number of replicas matches the desired number, because the controller has already created a new pod. It shows four running pods because a pod that's terminating is still considered running, although it isn't counted in the current replica count.
The list of events at the bottom shows the actions taken by the ReplicationController—it has created four pods so far.

UNDERSTANDING EXACTLY WHAT CAUSED THE CONTROLLER TO CREATE A NEW POD

The controller is responding to the deletion of a pod by creating a new replacement pod (see figure). Well, technically, it isn't responding to the deletion itself, but the resulting state—the inadequate number of pods.

While a ReplicationController is immediately notified about a pod being deleted (the API server allows clients to watch for changes to resources and resource lists), that's not what causes it to create a replacement pod. The notification triggers the controller to check the actual number of pods and take appropriate action.



Horizontally scaling pods

You've seen how ReplicationControllers make sure a specific number of pod instances is always running. Because it's incredibly simple to change the desired number of replicas, this also means scaling pods horizontally is trivial.

Scaling the number of pods up or down is as easy as changing the value of the replicas field in the ReplicationController resource. After the change, the ReplicationController will either see too many pods exist (when scaling down) and delete part of them, or see too few of them (when scaling up) and create additional pods.

SCALING UP A REPLICATIONCONTROLLER

Your ReplicationController has been keeping three instances of your pod running. You're going to scale that number up to 10 now.

```
$ kubectl scale rc kubia --replicas=10

$ kubectl get po
NAME          READY    STATUS    RESTARTS    AGE
kubia-5r9sd   1/1      Running   0           17s
kubia-9z8qm   1/1      Running   0           25m
kubia-cw52j   1/1      Running   0           17s
```

```
kubia-h2g8c   1/1     Running   0          17s
kubia-nhnqv   1/1     Running   0          17s
kubia-t7zll   1/1     Running   0          17s
kubia-wjlhg   1/1     Running   0          28m
kubia-xvz99   1/1     Running   0          17s
kubia-xx2v2   1/1     Running   0          28m
kubia-z6nq8   1/1     Running   0          17s
```

SCALING A REPLICATIONCONTROLLER BY EDITING ITS DEFINITION

Instead of using the kubectl scale command, you're going to scale it in a declarative way by editing the ReplicationController's definition:

```
$ kubectl edit rc kubia
replicationcontroller/kubia edited
```

Set the value of
Spec:
  Replicas: 3

```
kubectl get po
NAME          READY   STATUS        RESTARTS   AGE
kubia-5r9sd   1/1     Terminating   0          3m11s
kubia-9z8qm   1/1     Running       0          28m
kubia-cw52j   1/1     Terminating   0          3m11s
kubia-h2g8c   1/1     Terminating   0          3m11s
kubia-nhnqv   1/1     Terminating   0          3m11s
kubia-t7zll   1/1     Terminating   0          3m11s
kubia-wjlhg   1/1     Running       0          31m
kubia-xvz99   1/1     Terminating   0          3m11s
kubia-xx2v2   1/1     Running       0          31m
kubia-z6nq8   1/1     Terminating   0          3m11s
```

UNDERSTANDING THE DECLARATIVE APPROACH TO SCALING

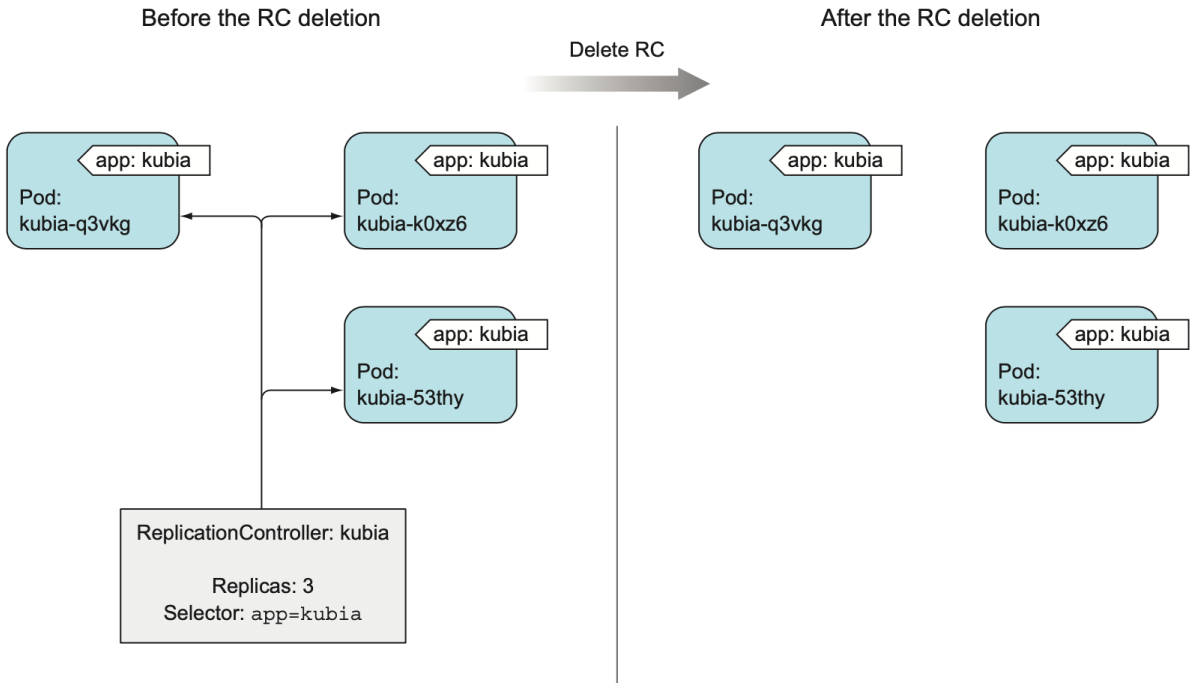
Horizontally scaling pods in Kubernetes is a matter of stating your desire: "I want to have *x* number of instances running." You're not telling Kubernetes what or how to do it. You're just specifying the desired state.

Deleting a ReplicationController

When you delete a ReplicationController through kubectl delete, the pods are also deleted. But because pods created by a ReplicationController aren't an integral part of the ReplicationController, and are only managed by it, you can delete only the ReplicationController and leave the pods running This may be useful when you initially have a set of pods managed by a ReplicationController, and then decide to replace the ReplicationController with a ReplicaSet, for example.

When deleting a ReplicationController with kubectl delete, you can keep its pods running by passing the --cascade=false option to the command. Try that now:

```
$ kubectl delete rc kubia --cascade=false
replicationcontroller "kubia" deleted
luca@MacBook-Pro-di-luca ~/D/0/U/k/replication> kubectl get po
NAME          READY   STATUS    RESTARTS   AGE
kubia-9z8qm   1/1     Running   0          33m
kubia-wjlhg   1/1     Running   0          36m
kubia-xx2v2   1/1     Running   0          36m
```



Using ReplicaSets instead of ReplicationControllers

Initially, ReplicationControllers were the only Kubernetes component for replicating pods and rescheduling them when nodes failed. Later, a similar resource called a ReplicaSet was introduced. It's a new generation of ReplicationController and replaces it completely (ReplicationControllers will eventually be deprecated).

You could have started this chapter by creating a ReplicaSet instead of a ReplicationController, but I felt it would be a good idea to start with what was initially available in Kubernetes. Plus, you'll still see ReplicationControllers used in the wild, so it's good for you to know about them. That said, you should always create ReplicaSets instead of ReplicationControllers from now on. They're almost identical, so you shouldn't have any trouble using them instead.

You usually won't create them directly, but instead have them created automatically when you create the higher-level Deployment resource. In any case, you should understand ReplicaSets, so let's see how they differ from ReplicationControllers.

Comparing a ReplicaSet to a ReplicationController

A ReplicaSet behaves exactly like a ReplicationController, but it has more expressive pod selectors. Whereas a ReplicationController's label selector only allows matching pods that include a certain label, a ReplicaSet's selector also allows matching pods that lack a certain label or pods that include a certain label key, regardless of its value.

Also, for example, a single ReplicationController can't match pods with the label env=production and those with the label env=devel at the same time. It can only match either pods with the env=production label or pods with the env=devel label. But a single ReplicaSet can match both sets of pods and treat them as a single group.
Similarly, a ReplicationController can't match pods based merely on the presence of a label key, regardless of its value, whereas a ReplicaSet can. For example, a ReplicaSet can match all pods that include a label with the key env, whatever its actual value is (you can think of it as env=*).

Defining a ReplicaSet
You're going to create a ReplicaSet now to see how the orphaned pods that were created by your ReplicationController and then abandoned earlier can now be adopted by a ReplicaSet. First, you'll rewrite your ReplicationController into a ReplicaSet by creating a new file called kubia-replicaset.yaml with the contents in the following listing.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: kubia
spec:
  replicas: 3
  selector:
    matchLabels:
      app: kubia
  template:
    metadata:
      labels:
        app: kubia
    spec:
      containers:
      - name: kubia
        image: luksa/kubia
```

The only difference is in the selector. Instead of listing labels the pods need to have directly under the selector property, you're specifying them under selector .matchLabels. This is the simpler (and less expressive) way of defining label selectors in a ReplicaSet.
Because you still have three pods matching the app=kubia selector running from earlier, creating this ReplicaSet will not cause any new pods to be created. The ReplicaSet will take those existing three pods under its wing.

Creating and examining a ReplicaSet

Create the ReplicaSet from the YAML file with the **kubectl create** command. After that, you can examine the ReplicaSet with kubectl get and kubectl describe:

```
$ kubectl create -f kubia-replicaset.yaml
replicaset.apps/kubia created
$ kubectl get po
NAME          READY   STATUS    RESTARTS   AGE
kubia-9z8qm   1/1     Running   0          42m
kubia-wjlhg   1/1     Running   0          45m
kubia-xx2v2   1/1     Running   0          45m

$ kubectl describe rs
Name:         kubia
Namespace:    default
Selector:     app=kubia
Labels:       <none>
Annotations:  <none>
Replicas:     3 current / 3 desired
Pods Status:  3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=kubia
  Containers:
   kubia:
    Image:        luksa/kubia
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:           <none>
```

As you can see, the ReplicaSet isn't any different from a ReplicationController. It's showing it has three replicas matching the selector. If you list all the pods, you'll see they're still the same three pods you had before. The ReplicaSet didn't create any new ones.

Using the ReplicaSet's more expressive label selectors

The main improvements of ReplicaSets over ReplicationControllers are their more expressive label selectors. You intentionally used the simpler matchLabels selector in the first ReplicaSet example to see that ReplicaSets are no different from Replication-

Controllers. Now, you'll rewrite the selector to use the more powerful matchExpressions property, as shown in the following listing.

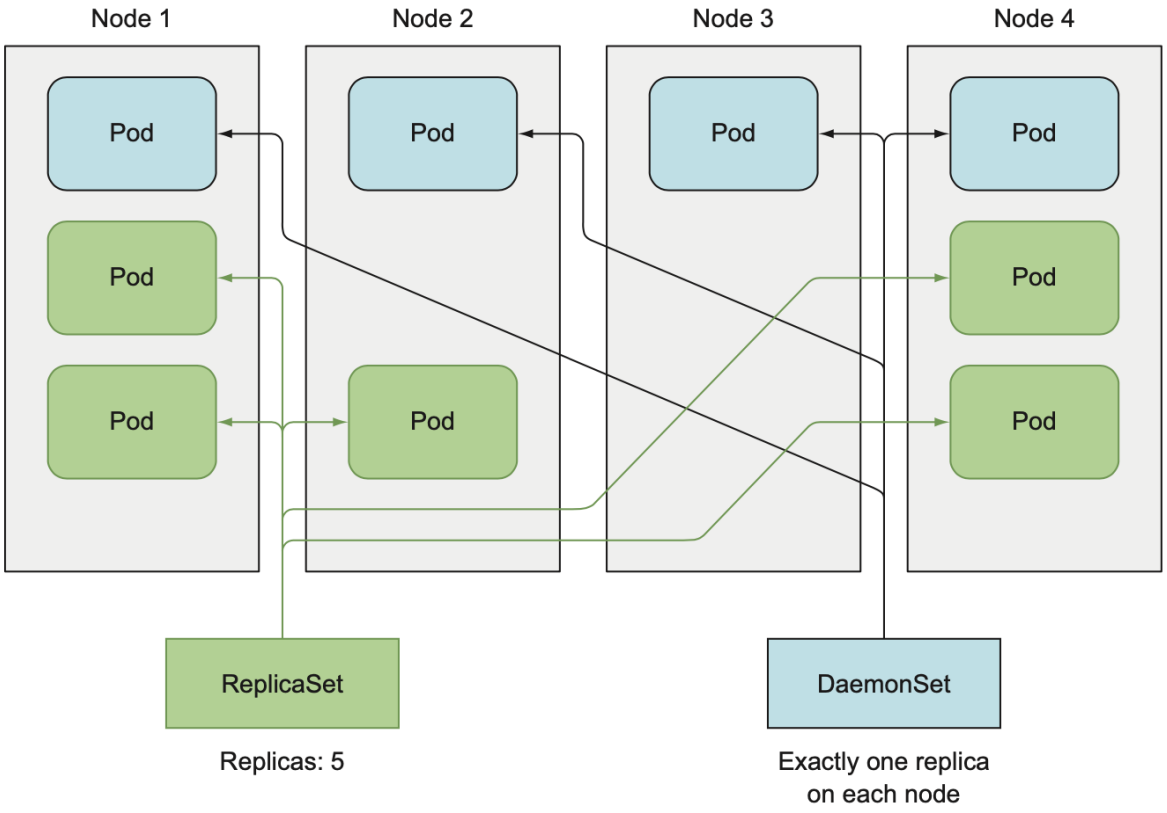
```
selector:
  matchExpressions:
    - key: app              #A
      operator: In          #B
      values:               #B
        - kubia                   #B
```

#A **This selector requires the pod to contain a label with the "app" key**
#B **The label's value must be "kubia".**

Running exactly one pod on each node with DaemonSets

Both ReplicationControllers and ReplicaSets are used for running a specific number of pods deployed anywhere in the Kubernetes cluster. But certain cases exist when you want a pod to run on each and every node in the cluster (and each node needs to run exactly one instance of the pod, as shown in figure 4.8).

Those cases include infrastructure-related pods that perform system-level opera- tions. For example, you'll want to run a log collector and a resource monitor on every node. Another good example is Kubernetes' own kube-proxy process, which needs to run on all nodes to make services work.



Outside of Kubernetes, such processes would usually be started through system init scripts or the systemd daemon during node boot up. On Kubernetes nodes, you can still use systemd to run your system processes, but then you can't take advantage of all the features Kubernetes provides.

Using a DaemonSet to run a pod on every node

To run a pod on all cluster nodes, you create a DaemonSet object, which is much like a ReplicationController or a ReplicaSet, except that pods created by a Daemon- Set already have a target node specified and skip the Kubernetes Scheduler. They aren't scattered around the cluster randomly.

A DaemonSet makes sure it creates as many pods as there are nodes and deploys each one on its own node, as shown in figure 4.8.

Whereas a ReplicaSet (or ReplicationController) makes sure that a desired num- ber of pod replicas exist in the cluster, a DaemonSet doesn't have any notion of a desired replica count. It doesn't need it because its job is to ensure that a pod match- ing its pod selector is running on each node.

If a node goes down, the DaemonSet doesn't cause the pod to be created else- where. But when a new node is added to the cluster, the DaemonSet immediately deploys a new pod instance to it. It also does the same if someone inadvertently deletes one of the pods, leaving the node without the DaemonSet's pod. Like a Replica- Set, a DaemonSet creates the pod from the pod template configured in it.

Using a DaemonSet to run pods only on certain nodes

A DaemonSet deploys pods to all nodes in the cluster, unless you specify that the pods should only run on a subset of all the nodes. This is done by specifying the node- Selector property in the pod template, which is part of the DaemonSet definition (similar to the pod template in a ReplicaSet or ReplicationController).

You've already used node selectors to deploy a pod onto specific nodes in chapter 3. A node selector in a DaemonSet is similar—it defines the nodes the DaemonSet must deploy its pods to.

EXPLAINING DAEMONSETS WITH AN EXAMPLE

Let's imagine having a daemon called ssd-monitor that needs to run on all nodes that contain a solid-state drive (SSD). You'll create a DaemonSet that runs this dae- mon on all nodes that are marked as having an SSD. The cluster administrators have added the disk=ssd label to all such nodes, so you'll create the DaemonSet with a node selector that only selects nodes with that label, as shown in figure 4.9.

CREATING A DAEMONSET YAML DEFINITION

You'll create a DaemonSet that runs a mock ssd-monitor process, which prints "SSD OK" to the standard output every five seconds. I've already prepared the mock container image and pushed it to Docker Hub, so you can use it instead of building your own. Create the YAML for the DaemonSet, as shown in the following listing.

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: ssd-monitor
spec:
  selector:
    matchLabels:
      app: ssd-monitor
  template:
    metadata:
      labels:
        app: ssd-monitor
    spec:                  #A
      nodeSelector:
        disk: ssd
      containers:
      - name: main
```

```
        image: luksa/ssd-monitor
```

#A The pod template includes a node selector, which selects nodes with the disk=ssd label.

You're defining a DaemonSet that will run a pod with a single container based on the luksa/ssd-monitor container image. An instance of this pod will be created for each node that has the disk=ssd label.

CREATING THE DAEMONSET
You'll create the DaemonSet like you always create resources from a YAML file:

```
$ kubectl create -f ssd-monitor-daemonset.yaml
daemonset "ssd-monitor" created
```

Let's see the created DaemonSet:

```
$ kubectl get ds
NAME          DESIRED  CURRENT  READY  UP-TO-DATE  AVAILABLE
NODE-SELECTOR
ssd-monitor   0        0        0      0           0
disk=ssd
```

Those zeroes look strange. Didn't the DaemonSet deploy any pods? List the pods:

```
$ kubectl get po
No resources found.
```

Where are the pods? Do you know what's going on? Yes, you forgot to label your nodes with the disk=ssd label. No problem—you can do that now. The DaemonSet should detect that the nodes' labels have changed and deploy the pod to all nodes with a matching label. Let's see if that's true.

ADDING THE REQUIRED LABEL TO YOUR NODE(S)
Regardless if you're using Minikube, GKE, or another multi-node cluster, you'll need to list the nodes first, because you'll need to know the node's name when labeling it:

```
$ kubectl get node
NAME       STATUS    AGE       VERSION
minikube   Ready     4d        v1.6.0
```

Now, add the disk=ssd label to one of your nodes like this:

```
$ kubectl label node minikube disk=ssd
node "minikube" labeled
```

The DaemonSet should have created one pod now. Let's see:

```
$ kubectl get po
NAME                READY     STATUS    RESTARTS   AGE
ssd-monitor-hgxwq   1/1       Running   0          35s
```

Okay; so far so good. If you have multiple nodes and you add the same label to further nodes, you'll see the DaemonSet spin up pods for each of them.

REMOVING THE REQUIRED LABEL FROM THE NODE

Now, imagine you've made a mistake and have mislabeled one of the nodes. It has a spinning disk drive, not an SSD. What happens if you change the node's label?

```
$ kubectl label node minikube disk=hdd --overwrite
```
node "minikube" labeled

Let's see if the change has any effect on the pod that was running on that node:

```
$ kubectl get po
NAME                READY     STATUS        RESTARTS   AGE
ssd-monitor-hgxwq   1/1       Terminating   0          4m
```

The pod is being terminated. But you knew that was going to happen, right? This wraps up your exploration of DaemonSets, so you may want to delete your ssd-monitor DaemonSet. If you still have any other daemon pods running, you'll see that deleting the DaemonSet deletes those pods as well.

You've learned about pods and how to deploy them through ReplicaSets and similar resources to ensure they keep running. Although certain pods can do their work independently of an external stimulus, many applications these days are meant to respond to external requests. For example, in the case of microservices, pods will usually respond to HTTP requests coming either from other pods inside the cluster or from clients outside the cluster.
Pods need a way of finding other pods if they want to consume the services they provide. Unlike in the non-Kubernetes world, where a sysadmin would configure each client app by specifying the exact IP address or hostname of the server providing the service in the client's configuration files, doing the same in Kubernetes wouldn't work, because

- *Pods are ephemeral*—They may come and go at any time, whether it's because a pod is removed from a node to make room for other pods, because someone scaled down the number of pods, or because a cluster node has failed.
- *Kubernetes assigns an IP address to a pod after the pod has been scheduled to a node and before it's started*—Clients thus can't know the IP address of the server pod up front.
- *Horizontal scaling means multiple pods may provide the same service*—Each of those pods has its own IP address. Clients shouldn't care how many pods are backing the service and what their IPs are. They shouldn't have to keep a list of all the individual IPs of pods. Instead, all those pods should be accessible through a single IP address.

To solve these problems, Kubernetes also provides another resource type—Services

Introducing services

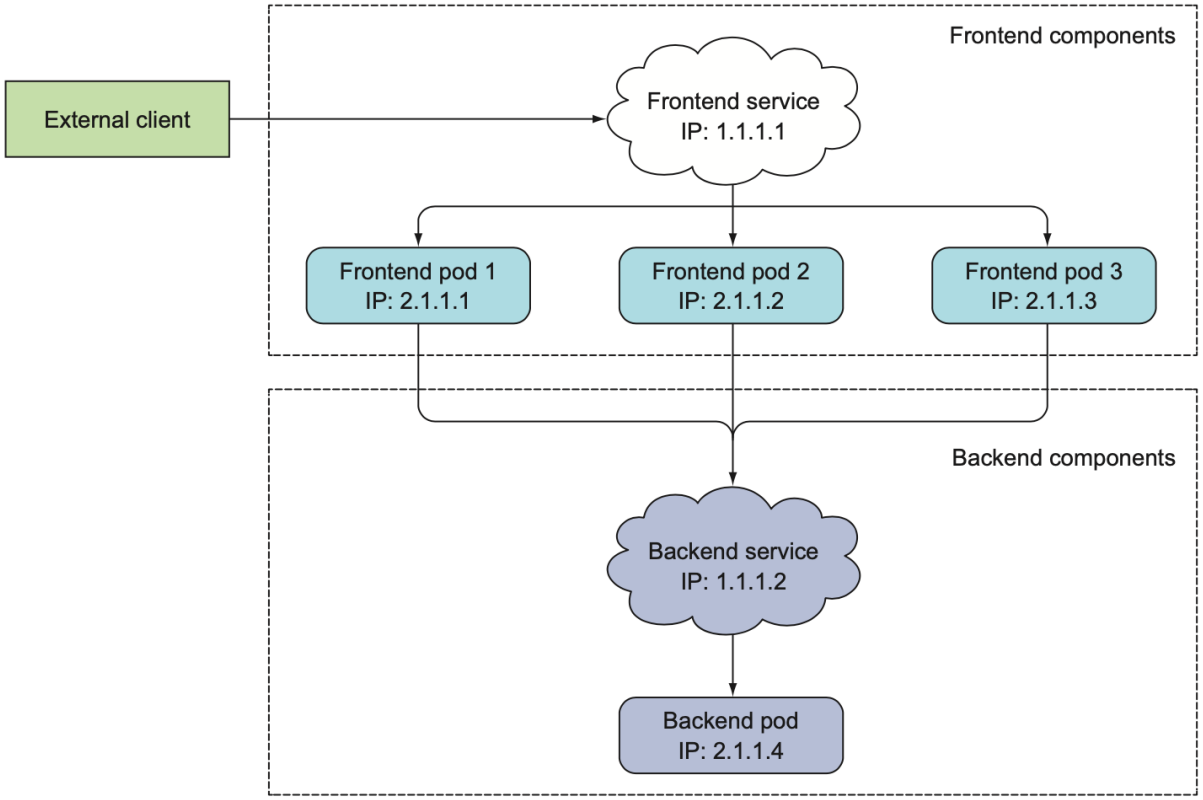
A Kubernetes Service is a resource you create to make a single, constant point of entry to a group of pods providing the same service. Each service has an IP address and port that never change while the service exists. Clients can open connections to that IP and port, and those connections are then routed to one of the pods backing that service. This way, clients of a service don't need to know the location of individual pods providing the service, allowing those pods to be moved around the cluster at any time.

EXPLAINING SERVICES WITH AN EXAMPLE

Let's revisit the example where you have a frontend web server and a backend database server. There may be multiple pods that all act as the frontend, but there may only be a single backend database pod. You need to solve two problems to make the system function:

- External clients need to connect to the frontend pods without caring if there's only a single web server or hundreds.
- The frontend pods need to connect to the backend database. Because the database runs inside a pod, it may be moved around the cluster over time, causing its IP address to change. You don't want to reconfigure the frontend pods every time the backend database is moved.

By creating a service for the frontend pods and configuring it to be accessible from outside the cluster, you expose a single, constant IP address through which external clients can connect to the pods. Similarly, by also creating a service for the backend pod, you create a stable address for the backend pod. The service address doesn't change even if the pod's IP address changes. Additionally, by creating the service, you also enable the frontend pods to easily find the backend service by its name through either environment variables or DNS. All the components of your system (the two services, the two sets of pods backing those services, and the interdependencies between them) are shown in figure.



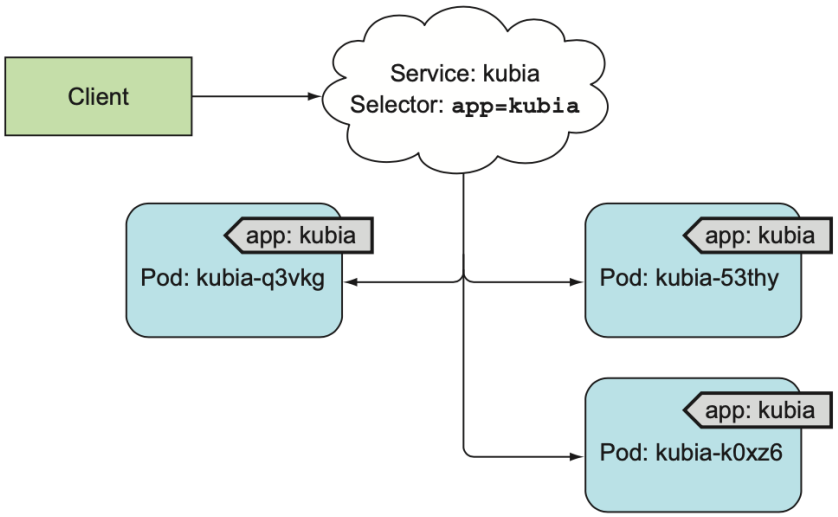
You now understand the basic idea behind services. Now, let's dig deeper by first seeing how they can be created.

Creating services

As you've seen, a service can be backed by more than one pod. Connections to the service are load-balanced across all the backing pods. But how exactly do you define which pods are part of the service and which aren't?

You probably remember label selectors and how they're used in ReplicationControllers and other pod controllers to specify which pods belong to the same set. The same mechanism is used by services in the same way, as you can see in figure.

In the previous chapter, you created a ReplicationController which then ran three instances of the pod containing the Node.js app. Create the ReplicationController again and verify three pod instances are up and running. After that, you'll create a Service for those three pods.



CREATING A SERVICE THROUGH KUBECTL EXPOSE

The easiest way to create a service is through **kubectl expose**, which you've already to expose the ReplicationController you created earlier. The expose command created a Service resource with the same pod selector as the one used by the ReplicationController, thereby exposing all its pods through a single IP address and port.

Now, instead of using the expose command, you'll create a service manually by posting a YAML to the Kubernetes API server.

CREATING A SERVICE THROUGH A YAML DESCRIPTOR
Create a file called kubia-svc.yaml with the following listing's contents.

```
apiVersion: v1
kind: Service
metadata:
  name: kubia
spec:
  ports:
  - port: 80          #A
    targetPort: 8080  #B
  selector:           #C
    app: kubia
```

#A The port this service will be available on
#B The container port the service will forward to
#C All pods with the app=kubia label will be part of this service.

You're defining a service called kubia, which will accept connections on port 80 and route each connection to port 8080 of one of the pods matching the app=kubia label selector. Go ahead and create the service by posting the file using **kubectl create**.

EXAMINING YOUR NEW SERVICE
After posting the YAML, you can list all Service resources in your namespace and see that an internal cluster IP has been assigned to your service:

```
$ kubectl get service
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP   68d
kubia        ClusterIP   10.102.158.76   <none>        80/TCP    16s
```

The list shows that the IP address assigned to the service is 10.102.158.193. Because this is the cluster IP, it's only accessible from inside the cluster. The primary purpose of services is exposing groups of pods to other pods in the cluster, but you'll usually also want to expose services externally. You'll see how to do that later. For now, let's use your service from inside the cluster and see what it does.

TESTING YOUR SERVICE FROM WITHIN THE CLUSTER
You can send requests to your service from within the cluster in a few ways:

- The obvious way is to create a pod that will send the request to the service's cluster IP and log the response. You can then examine the pod's log to see what the service's response was.
- You can ssh into one of the Kubernetes nodes and use the curl command.
- You can execute the curl command inside one of your existing pods through the kubectl exec command.

Let’s go for the last option, so you also learn how to run commands in existing pods.

REMOTELY EXECUTING COMMANDS IN RUNNING CONTAINERS

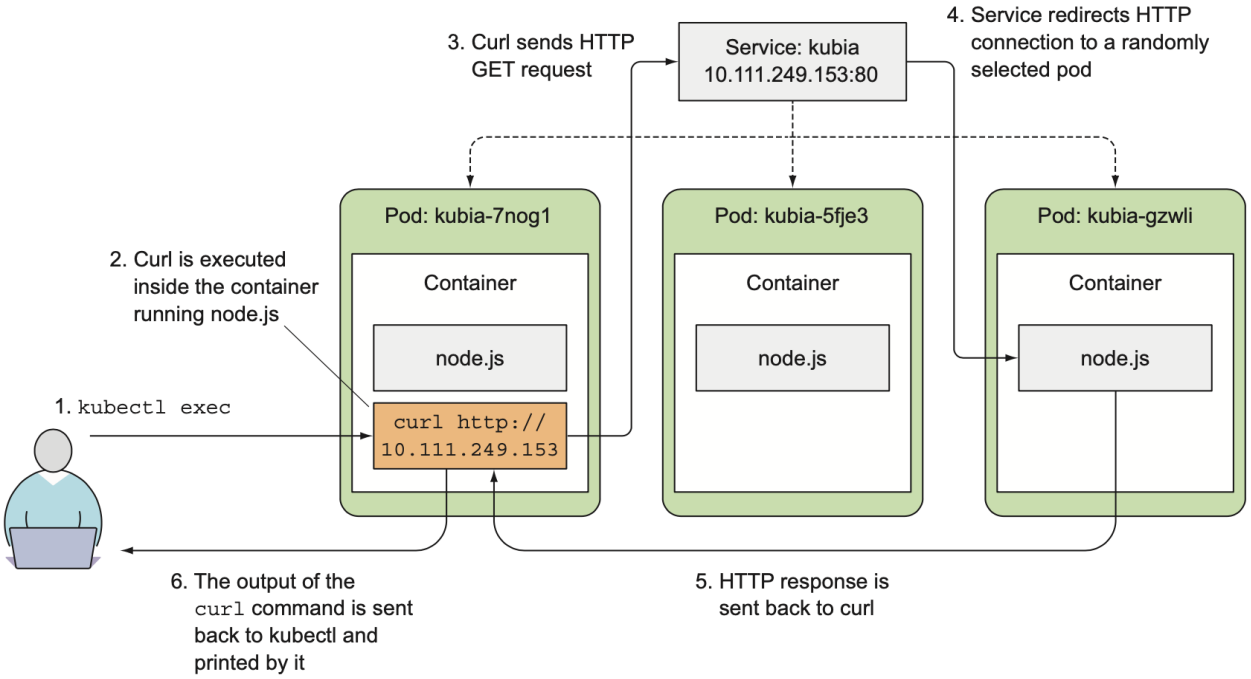
The kubectl exec command allows you to remotely run arbitrary commands inside an existing container of a pod. This comes in handy when you want to examine the contents, state, and/or environment of a container. List the pods with the kubectl get pods command and choose one as your target for the exec command (in the fol- lowing example, I’ve chosen the kubia-hst84 pod as the target). You’ll also need to obtain the cluster IP of your service (using kubectl get svc, for example). When running the following commands yourself, be sure to replace the pod name and the service IP with your own:

```
$ kubectl get pod
NAME          READY   STATUS    RESTARTS   AGE
kubia-b2v5k   1/1     Running   0          57m
kubia-hst84   1/1     Running   0          57m
kubia-l55vf   1/1     Running   0          57m

$ kubectl exec kubia-hst84 -- curl -s http://10.102.158.76
You've hit kubia-l55vf
```

If you’ve used docker to execute commands on a remote system before, you’ll recognize that kubectl exec isn’t much different.

Let’s go over what transpired when you ran the command. Figure shows the sequence of events. You instructed Kubernetes to execute the curl command inside the container of one of your pods. Curl sent an HTTP request to the service IP, which is backed by three pods. The Kubernetes service proxy intercepted the connection, selected a random pod among the three pods, and forwarded the request to it. Node.js running inside that pod then handled the request and returned an HTTP response containing the pod’s name. Curl then printed the response to the standard output, which was intercepted and printed to its standard output on your local machine by kubectl.



DISCOVERING SERVICES THROUGH DNS

Remember in the previous chapter when you listed pods in the kube-system namespace? One of the pods was called kube-dns. The kube-system namespace also includes a corresponding service with the same name.

As the name suggests, the pod runs a DNS server, which all other pods running in the cluster are automatically configured to use (Kubernetes does that by modifying each container's /etc/resolv.conf file). Any DNS query performed by a process running in a pod will be handled by Kubernetes' own DNS server, which knows all the services running in your system.

Each service gets a DNS entry in the internal DNS server, and client pods that know the name of the service can access it through its fully qualified domain name (FQDN) .

CONNECTING TO THE SERVICE THROUGH ITS FQDN

To revisit the frontend-backend example, a frontend pod can connect to the backend-database service by opening a connection to the following FQDN:

backend-database.default.svc.cluster.local

backend-database corresponds to the service name, default stands for the namespace the service is defined in, and svc.cluster.local is a configurable cluster domain suffix used in all cluster local service names.

Connecting to a service can be even simpler than that. You can omit the svc.cluster.local suffix and even the namespace, when the frontend pod is in the same namespace as the database pod. You can thus refer to the service simply as backend-database. That's incredibly simple, right?

RUNNING A SHELL IN A POD'S CONTAINER

You can use the kubectl exec command to run bash (or any other shell) inside a pod's container. This way you're free to explore the container as long as you want, without having to perform a kubectl exec for every command you want to run.

To use the shell properly, you need to pass the -it option to kubectl exec:

```
$ kubectl get po
NAME          READY   STATUS    RESTARTS   AGE
kubia-b2v5k   1/1     Running   0          74m
kubia-hst84   1/1     Running   0          74m
kubia-l55vf   1/1     Running   0          74m

kubectl exec -it kubia-b2v5k bash
kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in
a future version. Use kubectl exec [POD] -- [COMMAND] instead.
root@kubia-b2v5k:/# curl http://kubia.default.svc.cluster.local
You've hit kubia-l55vf
root@kubia-b2v5k:/# curl http://kubia.default
You've hit kubia-hst84
```

You can hit your service by using the service's name as the hostname in the requested URL. You can omit the namespace and the svc.cluster.local suffix because of how the DNS resolver inside each pod's container is configured. Look at the /etc/resolv.conf file in the container and you'll understand:

```
root@kubia-b2v5k:/# cat /etc/resolv.conf
nameserver 10.96.0.10
search default.svc.cluster.local svc.cluster.local cluster.local
```

Service endpoints

Before going into how to do this, let me first shed more light on services. Services don't link to pods directly. Instead, a resource sits in between—the Endpoints resource. You may have already noticed endpoints if you used the kubectl describe command on your service, as shown in the following listing.

```
$ kubectl describe svc kubia
Name:              kubia
Namespace:         default
Labels:            <none>
Annotations:       <none>
Selector:          app=kubia                                    #A
Type:              ClusterIP
IP:                10.102.158.76
Port:              <unset>  80/TCP
TargetPort:        8080/TCP
Endpoints:         172.17.0.2:8080,172.17.0.3:8080,172.17.0.4:8080 #B
Session Affinity:  None
Events:            <none>
```

#A The service's pod selector is used to create the list of endpoints.
#B The list of pod IPs and ports that represent the endpoints of this service

An Endpoints resource (yes, plural) is a list of IP addresses and ports exposing a service. The Endpoints resource is like any other Kubernetes resource, so you can display its basic info with kubectl get:

```
$ kubectl get endpoints kubia
NAME    ENDPOINTS                                         AGE
kubia   172.17.0.2:8080,172.17.0.3:8080,172.17.0.4:8080   91m
```

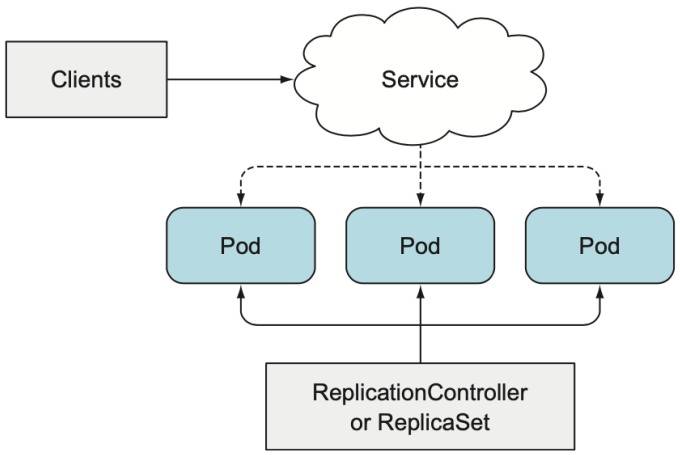
Although the pod selector is defined in the service spec, it's not used directly when redirecting incoming connections. Instead, the selector is used to build a list of IPs and ports, which is then stored in the Endpoints resource. When a client connects to a service, the service proxy selects one of those IP and port pairs and redirects the incoming connection to the server listening at that location.

Deployments

If you're going to want to update your app. This chapter covers how to update apps running in a Kubernetes cluster and how Kubernetes helps you move toward a true zero-downtime update process. Although this can be achieved using only ReplicationControllers or ReplicaSets, Kubernetes also provides a Deployment resource that sits on top of ReplicaSets and enables declarative application updates. If you're not completely sure what that means, keep reading—it's not as complicated as it sounds.

Updating applications running in pods

Let's start off with a simple example. Imagine having a set of pod instances providing a service to other pods and/or external clients. After reading this book up to this point, you likely recognize that these pods are backed by a ReplicationController or a ReplicaSet. A Service also exists through which clients (apps running in other pods or external clients) access the pods. This is how a basic application looks in Kubernetes (shown in figure).



Initially, the pods run the first version of your application—let's suppose its image is tagged as v1. You then develop a newer version of the app and push it to an image repository as a new image, tagged as v2. You'd next like to replace all the pods with this new version. Because you can't change an existing pod's image after the pod is created, you need to remove the old pods and replace them with new ones running the new image.
You have two ways of updating all those pods. You can do one of the following:

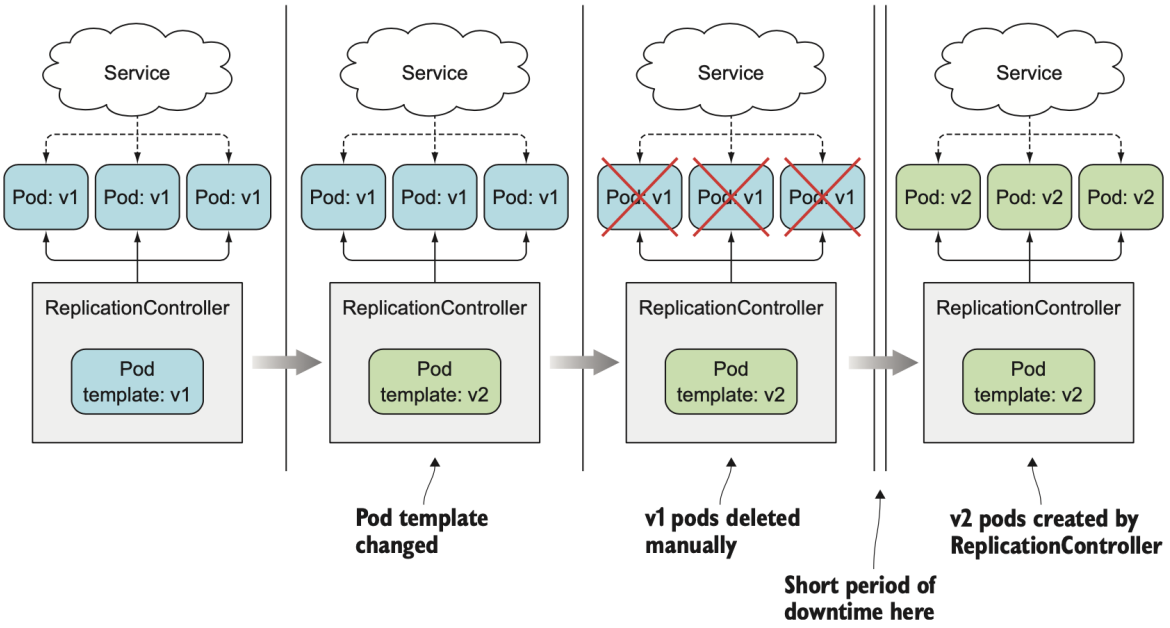
- Delete all existing pods first and then start the new ones.
- Start new ones and, once they're up, delete the old ones. You can do this either by adding all the new pods and then deleting all the old ones at once, or sequentially, by adding new pods and removing old ones gradually.

Both these strategies have their benefits and drawbacks. The first option would lead to a short period of time when your application is unavailable. The second option requires your app to handle running two versions of the app at the same time. If your app stores data in a

data store, the new version shouldn't modify the data schema or the data in such a way that breaks the previous version.
How do you perform these two update methods in Kubernetes? First, let's look at how to do this manually; then, once you know what's involved in the process, you'll learn how to have Kubernetes perform the update automatically.

Deleting old pods and replacing them with new ones

You already know how to get a ReplicationController to replace all its pod instances with pods running a new version. You probably remember the pod template of a ReplicationController can be updated at any time. When the ReplicationController creates new instances, it uses the updated pod template to create them.
If you have a ReplicationController managing a set of v1 pods, you can easily replace them by modifying the pod template so it refers to version v2 of the image and then deleting the old pod instances. The ReplicationController will notice that no pods match its label selector and it will spin up new instances. The whole process is shown in figure



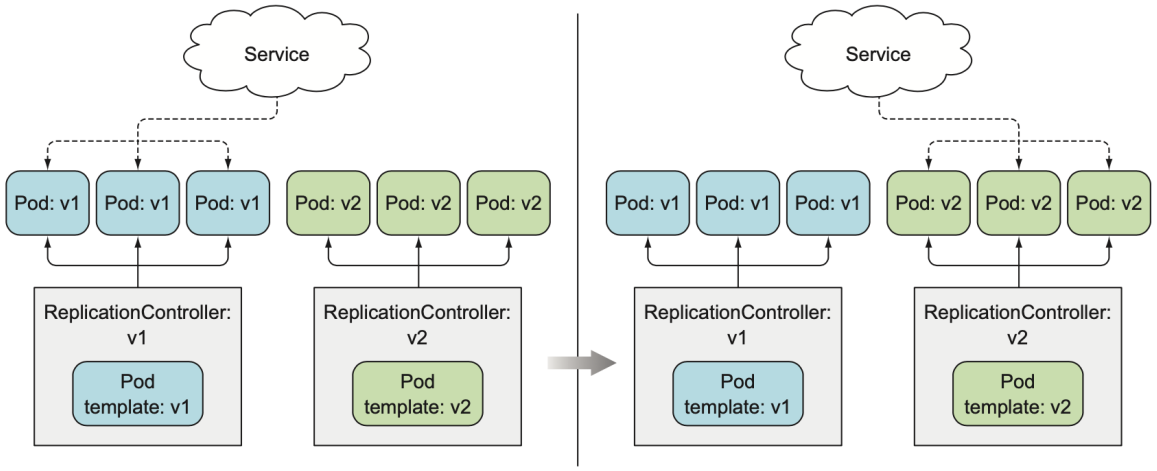
This is the simplest way to update a set of pods, if you can accept the short downtime between the time the old pods are deleted and new ones are started.

Spinning up new pods and then deleting the old ones

If you don't want to see any downtime and your app supports running multiple versions at once, you can turn the process around and first spin up all the new pods and only then delete the old ones. This will require more hardware resources, because you'll have double the number of pods running at the same time for a short while.
This is a slightly more complex method compared to the previous one, but you should be able to do it by combining what you've learned about ReplicationControllers and Services so far.

SWITCHING FROM THE OLD TO THE NEW VERSION AT ONCE

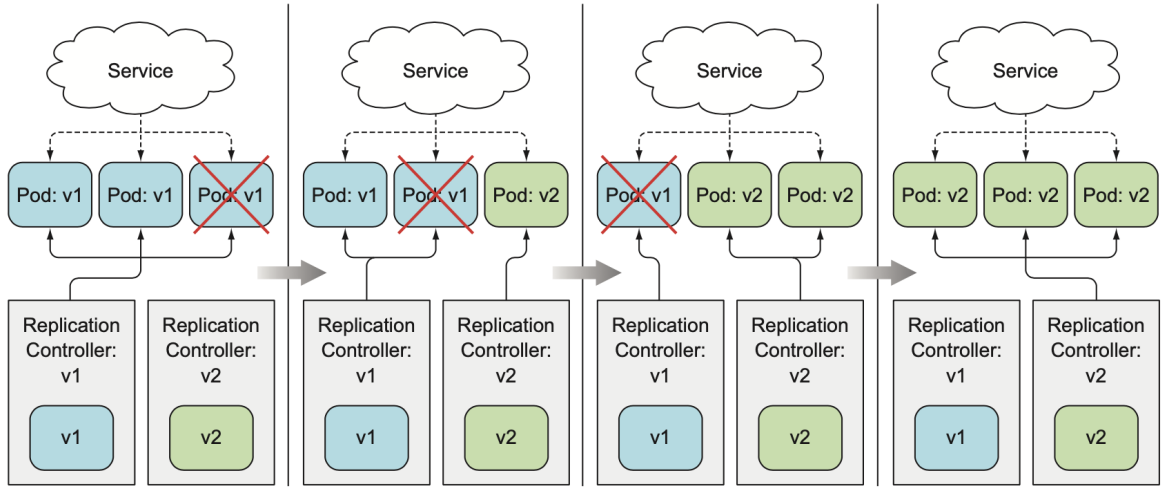
Pods are usually fronted by a Service. It's possible to have the Service front only the initial version of the pods while you bring up the pods running the new version. Then, once all the new pods are up, you can change the Service's label selector and have the Service switch over to the new pods, as shown in figure. This is called a *blue-green deployment*. After switching over, and once you're sure the new version functions correctly, you're free to delete the old pods by deleting the old ReplicationController.



PERFORMING A ROLLING UPDATE

Instead of bringing up all the new pods and deleting the old pods at once, you can also perform a rolling update, which replaces pods step by step. You do this by slowly scaling down the previous ReplicationController and scaling up the new one. In this case, you'll want the Service's pod selector to include both the old and the new pods, so it directs requests toward both sets of pods. See figure.

Doing a rolling update manually is laborious and error-prone. Depending on the number of replicas, you'd need to run a dozen or more commands in the proper order to perform the update process. Luckily, Kubernetes allows you to perform the rolling update with a single command. You'll learn how in the next section.



Using Deployments for updating apps declaratively

A Deployment is a higher-level resource meant for deploying applications and updating them declaratively, instead of doing it through a ReplicationController or a ReplicaSet, which are both considered lower-level concepts.

When you create a Deployment, a ReplicaSet resource is created underneath (eventually more of them). As you may remember from chapter 4, ReplicaSets are a new generation of ReplicationControllers, and should be used instead of them. ReplicaSets replicate and manage pods, as well. When using a Deployment, the actual pods are created and managed by the Deployment's ReplicaSets, not by the Deployment directly (the relationship is shown in figure).

You might wonder why you'd want to complicate things by introducing another object on top of a ReplicationController or ReplicaSet, when they're what suffices to keep a set of pod instances running. As the rolling update example in section demonstrates, when updating the app, you need to introduce an additional ReplicationController and coordinate the two controllers to dance around each other without stepping on each other's toes. You need something coordinating this dance. A Deployment resource takes care of that (it's not the Deployment resource itself, but the controller process running in the Kubernetes control plane that does that).

Using a Deployment instead of the lower-level constructs makes updating an app much easier, because you're defining the desired state through the single Deployment resource and letting Kubernetes take care of the rest, as you'll see in the next few pages.

Running the initial version of the app

Obviously, before you can update an app, you need to have an app deployed. You're going to use a slightly modified version of the kubia NodeJS app you as your initial version. It's a simple webapp that returns the pod's hostname in the HTTP response.

CREATING THE V1 APP

You'll change the app so it also returns its version number in the response, which will allow you to distinguish between the different versions you're about to build.

The v1 version of our app: v1/app.js

```
const http = require('http');
        const os = require('os');
        console.log("Kubia server starting...");
var handler = function(request, response) {
  console.log("Received request from " +
request.connection.remoteAddress);
  response.writeHead(200);
  response.end("This is v1 running in pod " + os.hostname() +
"\n");
};
var www = http.createServer(handler);
```

```
www.listen(8080);
```

The version 2 of the app. To keep things simple, all you'll do is change the response to say, "This is v2":

```
 response.end("This is v2 running in pod " + os.hostname() +
"\n");
```

Creating a Deployment

Creating a Deployment isn't that different from creating a ReplicationController. A Deployment is also composed of a label selector, a desired replica count, and a pod template. In addition to that, it also contains a field, which specifies a deployment strategy that defines how an update should be performed when the Deployment resource is modified.

CREATING A DEPLOYMENT MANIFEST

Let's see how to use the kubia-v1 ReplicationController example from earlier in this chapter and modify it so it describes a Deployment instead of a ReplicationController. As you'll see, this requires only three trivial changes. The following listing shows the modified YAML.

kubia-deployment-v1.yaml:

```
apiVersion: apps/v1        #A
kind: Deployment                #B
metadata:
  name: kubia
  labels:
    app: kubia
spec:
  replicas: 3
  selector:
    matchLabels:
      app: kubia
  template:
    metadata:
      labels:
        app: kubia
    spec:
      containers:
      - name: nodejs
        image: luksa/kubia:v1
        ports:
        - containerPort: 8080
```

#A Deployments are in the apps API group, version v1
#B You've changed the kind from ReplicationController to Deployment.

CREATING THE DEPLOYMENT RESOURCE
Before you create this Deployment, make sure you delete any ReplicationControllers and pods that are still running, but keep the kubia Service for now. You can use the --all switch to delete all those ReplicationControllers like this:

```
$ kubectl delete rc --all
```

You're now ready to create the Deployment:

```
$ kubectl create -f kubia-deployment-v1.yaml --record
deployment.apps/kubia created
```

TIP Be sure to include the --record command-line option when creating it. This records the command in the revision history, which will be useful later.

DISPLAYING THE STATUS OF THE DEPLOYMENT ROLLOUT
You can use the usual kubectl get deployment and the kubectl describe deployment commands to see details of the Deployment, but let me point you to an additional command, which is made specifically for checking a Deployment's status:

```
$ kubectl rollout status deployment kubia
deployment "kubia" successfully rolled out
```

According to this, the Deployment has been successfully rolled out, so you should see the three pod replicas up and running. Let's see:

```
$ kubectl get po
NAME                     READY   STATUS    RESTARTS   AGE
kubia-5f78bc5dc5-4d6qv   1/1     Running   0          2m54s
kubia-5f78bc5dc5-5pqjp   1/1     Running   0          2m54s
kubia-5f78bc5dc5-q9p92   1/1     Running   0          2m54s
```

UNDERSTANDING HOW DEPLOYMENTS CREATE REPLICASETS, WHICH THEN CREATE THE PODS
Take note of the names of these pods. Earlier, when you used a ReplicationController to create pods, their names were composed of the name of the controller plus a randomly generated string (for example, kubia-v1-m33mv). The three pods created by the Deployment include an additional numeric value in the middle of their names. What is that exactly?
The number corresponds to the hashed value of the pod template in the Deployment and the ReplicaSet managing these pods. As we said earlier, a Deployment doesn't manage pods directly. Instead, it creates ReplicaSets and leaves the managing to them, so let's look at the ReplicaSet created by your Deployment:

```
$ kubectl get replicasets
NAME               DESIRED   CURRENT   READY   AGE
kubia-5f78bc5dc5   3         3         3       3m39s
```

The ReplicaSet's name also contains the hash value of its pod template. As you'll see later, a Deployment creates multiple ReplicaSets—one for each version of the pod template. Using the hash value of the pod template like this allows the Deployment to always use the same (possibly existing) ReplicaSet for a given version of the pod template.

SLOWING DOWN THE ROLLING UPDATE FOR DEMO PURPOSES

In the next exercise, you'll use the RollingUpdate strategy, but you need to slow down the update process a little, so you can see that the update is indeed performed in a rolling fashion. You can do that by setting the minReadySeconds attribute on the Deployment. We'll explain what this attribute does by the end of this chapter. For now, set it to 10 seconds with the kubectl patch command.

```
$ kubectl patch deployment kubia -p '{"spec": {"minReadySeconds":
10}}'
"kubia" patched
```

The kubectl patch command is useful for modifying a single property or a limited number of properties of a resource without having to edit its defi- nition in a text editor.

TRIGGERING THE ROLLING UPDATE

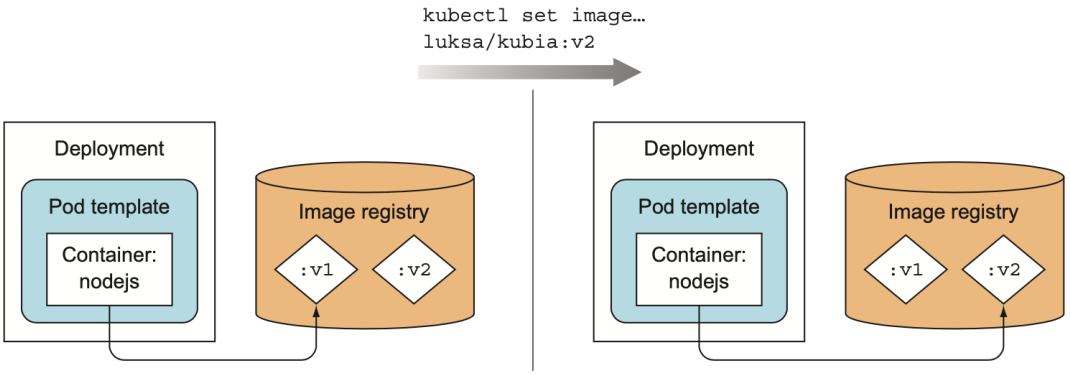
If you'd like to track the update process as it progresses, first run the curl loop in another pod (for example the nginx pod used in LAB1) to see what's happening with the requests :

```
root@nginx:/# while true;  do curl kubia.default; sleep 2; done;
```

To trigger the actual rollout, you'll change the image used in the single pod container to luksa/kubia:v2. Instead of editing the whole YAML of the Deployment object or using the patch command to change the image, you'll use the kubectl set image command, which allows changing the image of any resource that contains a container (ReplicationControllers, ReplicaSets, Deployments, and so on). You'll use it to modify your Deployment like this:

```
$ kubectl set image deployment kubia nodejs=luksa/kubia:v2
```

When you execute this command, you're updating the kubia Deployment's pod template so the image used in its nodejs container is changed to luksa/kubia:v2 (from :v1). This is shown in figure


kubectl set image…
luksa/kubia:v2
Deployment
Pod template
Container:
nodejs
Image registry
:v1
:v2
Deployment
Pod template
Container:
nodejs
Image registry
:v1
:v2

Ways of modifying Deployments and other resources

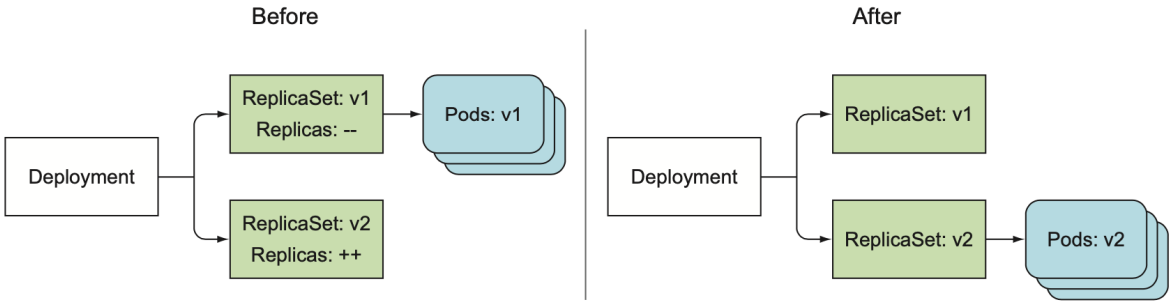
Method	What it does
kubectl edit	Opens the object's manifest in your default editor. After making changes, saving the file, and exiting the editor, the object is updated. Example: kubectl edit deployment kubia
kubectl patch	Modifies individual properties of an object. Example: kubectl patch deployment kubia -p '{"spec": {"template": {"spec": {"containers": [{"name": "nodejs", "image": "luksa/kubia:v2"}]}}}}'
kubectl apply	Modifies the object by applying property values from a full YAML or JSON file. If the object specified in the YAML/JSON doesn't exist yet, it's created. The file needs to contain the full definition of the resource (it can't include only the fields you want to update, as is the case with kubectl patch). Example: kubectl apply -f kubia-deployment-v2.yaml
kubectl replace	Replaces the object with a new one from a YAML/JSON file. In contrast to the apply command, this command requires the object to exist; otherwise it prints an error. Example: kubectl replace -f kubia-deployment-v2.yaml
kubectl set image	Changes the container image defined in a Pod, ReplicationController's template, Deployment, DaemonSet, Job, or ReplicaSet. Example: kubectl set image deployment kubia nodejs=luksa/kubia:v2

If you've run the curl loop, you'll see requests initially hitting only the v1 pods; then more and more of them hit the v2 pods until, finally, all of them hit only the remaining v2 pods, after all v1 pods are deleted.

UNDERSTANDING THE AWESOMENESS OF DEPLOYMENTS

Let's think about what has happened. By changing the pod template in your Deployment resource, you've updated your app to a newer version—by changing a single field!
The controllers running as part of the Kubernetes control plane then performed the update. The process wasn't performed by the kubectl client. I don't know about you, but I think that's simpler than having to run a special command telling Kubernetes what to do and then waiting around for the process to be completed.

An additional ReplicaSet was created and it was then scaled up slowly, while the previous ReplicaSet was scaled down to zero (the initial and final states are shown in figure).



You can still see the old ReplicaSet next to the new one if you list them:

```
$ kubectl get rs
NAME               DESIRED   CURRENT   READY   AGE
kubia-55d48448fd   3         3         3       11m
kubia-5f78bc5dc5   0         0         0       36m
```

Rolling back a deployment

You're currently running version v2 of your image, so you'll need to prepare version 3 first.

CREATING VERSION 3 OF YOUR APP

In version 3, you'll introduce a bug that makes your app handle only the first four requests properly. All requests from the fifth request onward will return an internal server error (HTTP status code 500). You'll simulate this by adding an if statement at the beginning of the handler function. The following listing shows the new code, with all required changes shown in bold.

v3/app.js

```
const http = require('http');
const os = require('os');
var requestCount = 0;
console.log("Kubia server starting...");
var handler = function(request, response) {
console.log("Received request from " +
request.connection.remoteAddress); if (++requestCount >= 5) {
    response.writeHead(500);
    response.end("Some internal error has occurred! This is pod " +
     os.hostname() + "\n");
    return;
}
  response.writeHead(200);
  response.end("This is v3 running in pod " + os.hostname() +
"\n");
};
var www = http.createServer(handler);
www.listen(8080);
```

As you can see, on the fifth and all subsequent requests, the code returns a 500 error with the message “Some internal error has occurred...”

DEPLOYING VERSION 3

We are now using a v3 version of the image available as luksa/kubia:v3. You’ll deploy this new version by changing the image in the Deployment specification again:

```
$ kubectl set image deployment kubia nodejs=luksa/kubia:v3
deployment "kubia" image updated
```

You can follow the progress of the rollout with kubectl rollout status:

```
$ kubectl rollout status deployment kubia
```

The new version is now live. As the following listing shows, after a few requests, your web clients start receiving errors.

```
Some internal error has occurred! This is pod
kubia-56fc5b76bf-wj6cf
Some internal error has occurred! This is pod
kubia-56fc5b76bf-npwvr
Some internal error has occurred! This is pod
kubia-56fc5b76bf-wj6cf
Some internal error has occurred! This is pod
kubia-56fc5b76bf-npwvr
Some internal error has occurred! This is pod
kubia-56fc5b76bf-npwvr
```

UNDOING A ROLLOUT

You can’t have your users experiencing internal server errors, so you need to do something about it fast. Let’s see what you can do about your bad rollout manually. Luckily, Deployments make it easy to roll back to the previously deployed version by telling Kubernetes to undo the last rollout of a Deployment:

```
$ kubectl rollout undo deployment kubia
deployment.apps/kubia rolled back
```

This rolls the Deployment back to the previous revision.

DISPLAYING A DEPLOYMENT’S ROLLOUT HISTORY

Rolling back a rollout is possible because Deployments keep a revision history. As you’ll see later, the history is stored in the underlying ReplicaSets. When a rollout completes, the old ReplicaSet isn’t deleted, and this enables rolling back to any revi- sion, not only the previous one. The revision history can be displayed with the kubectl rollout history command:

```
$ kubectl rollout history deployment kubia
deployment.apps/kubia
REVISION  CHANGE-CAUSE
1         kubectl create --filename=kubia-deployment-v1.yaml --record=true
```

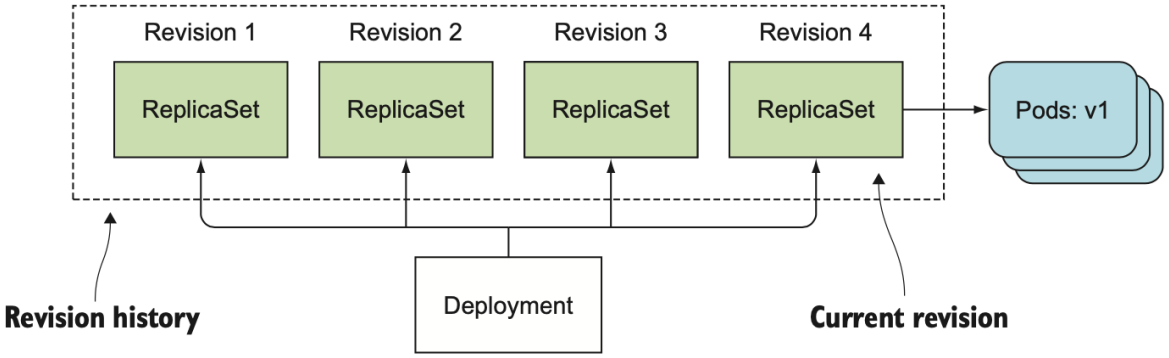
```
3          kubectl create --filename=kubia-deployment-v1.yaml --record=true
4          kubectl create --filename=kubia-deployment-v1.yaml --record=true
```

ROLLING BACK TO A SPECIFIC DEPLOYMENT REVISION

You can roll back to a specific revision by specifying the revision in the undo command. For example, if you want to roll back to the first version, you'd execute the following command:

```
$ kubectl rollout undo deployment kubia --to-revision=1
```

Remember the inactive ReplicaSet left over when you modified the Deployment the first time? The ReplicaSet represents the first revision of your Deployment. All ReplicaSets created by a Deployment represent the complete revision history, as shown in figure. Each ReplicaSet stores the complete information of the Deployment at that specific revision, so you shouldn't delete it manually. If you do, you'll lose that specific revision from the Deployment's history, preventing you from rolling back to it.



DEFINING A READINESS PROBE TO PREVENT OUR V3 VERSION FROM BEING ROLLED OUT FULLY

You're going to deploy version v3 again, but this time, you'll have the proper readi- ness probe defined on the pod. Your Deployment is currently at version v4, so before you start, roll back to version v2 again so you can pretend this is the first time you're upgrading to v3. If you wish, you can go straight from v4 to v3, but the text that fol- lows assumes you returned to v2 first.

Unlike before, where you only updated the image in the pod template, you're now also going to introduce a readiness probe for the container at the same time. Up until now, because there was no explicit readiness probe defined, the container and the pod were always considered ready, even if the app wasn't truly ready or was returning errors. There was no way for Kubernetes to know that the app was malfunctioning and shouldn't be exposed to clients.

To change the image and introduce the readiness probe at once, you'll use the kubectl apply command. You'll use the following YAML to update the deployment (you'll store it as kubia-deployment-v3-with-readinesscheck.yaml), as shown in the following listing.

kubia-deployment-v3-with-readinesscheck.yaml

```
apiVersion: apps/v1        #A
kind: Deployment                 #B
metadata:
  name: kubia
  labels:
    app: kubia
spec:
  replicas: 3
  minReadySeconds: 10
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
    type: RollingUpdate
  selector:
    matchLabels:
      app: kubia
  template:
    metadata:
      labels:
        app: kubia
    spec:
      containers:
     - name: nodejs
        image: luksa/kubia:v3
        readinessProbe:
          periodSeconds: 1
          httpGet:
            path: /
            port: 8080
        ports:
        - containerPort: 8080
```

UPDATING A DEPLOYMENT WITH KUBECTL APPLY
You're defining a readiness probe that will be executed every second.
The readiness probe will perform an HTTP GET request against our container.
To update the Deployment this time, you'll use kubectl apply like this:

The apply command updates the Deployment with everything that's defined in the YAML file. It not only updates the image but also adds the readiness probe definition and anything else you've added or modified in the YAML. If the new YAML also contains the replicas field, which doesn't match the number of replicas on the existing Deployment, the apply operation will also scale the Deployment, which isn't usually what you want.

Running the apply command will kick off the update process, which you can again follow with the rollout status command:

```
$ kubectl rollout status deployment kubia
Waiting for deployment "kubia" rollout to finish: 1 out of 3 new
replicas have been updated...
```

Because the status says one new pod has been created, your service should be hitting it occasionally, right? Let's see:

```
root@nginx:/# while true;  do curl kubia.default; sleep 2; done;
This is v2 running in pod kubia-55d48448fd-lv27q
This is v2 running in pod kubia-55d48448fd-tb2cc
This is v2 running in pod kubia-55d48448fd-tb2cc
This is v2 running in pod kubia-55d48448fd-lv27q
This is v2 running in pod kubia-55d48448fd-9862x
This is v2 running in pod kubia-55d48448fd-lv27q
```

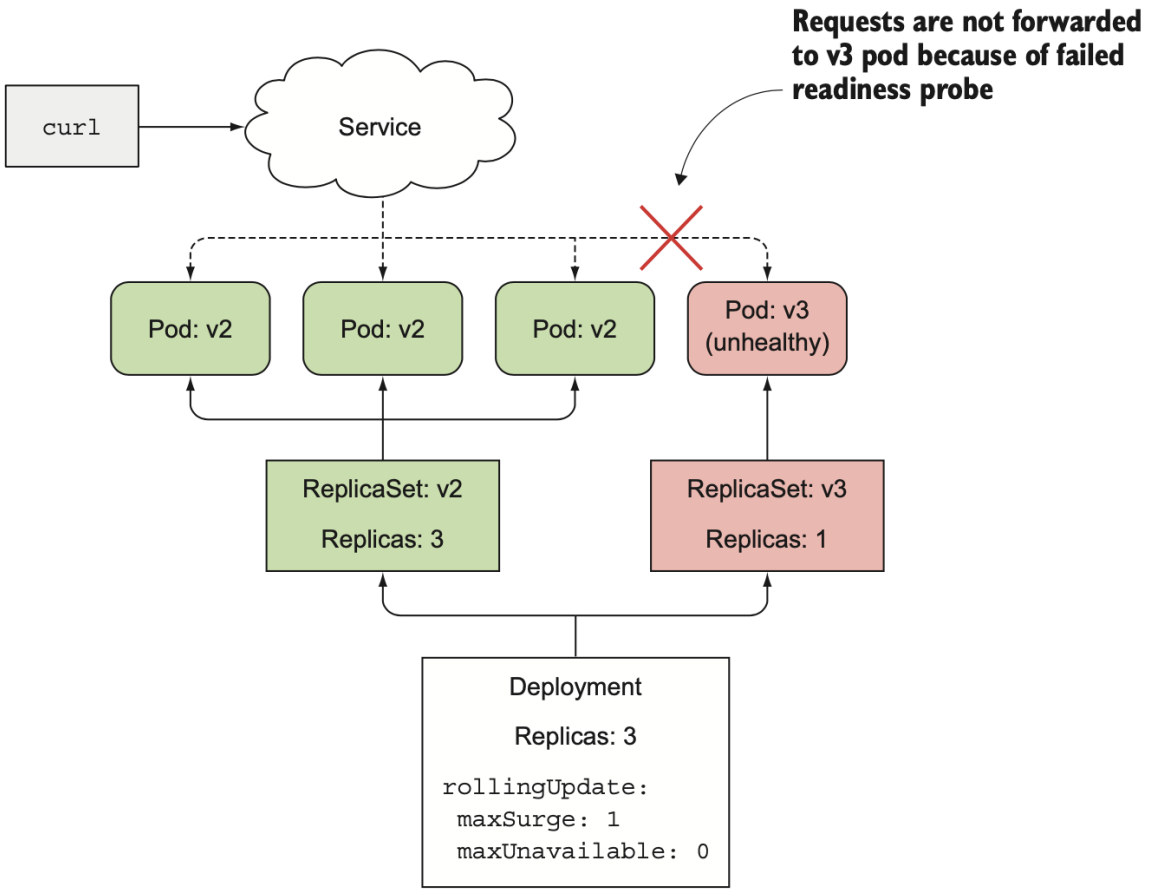
Nope, you never hit the v3 pod. Why not? Is it even there? List the pods:

```
$ kubectl get po
130
NAME                     READY   STATUS             RESTARTS   AGE
kubia-55d48448fd-9862x   1/1     Running            0          28m
kubia-55d48448fd-lv27q   1/1     Running            0          28m
kubia-55d48448fd-tb2cc   1/1     Running            0          28m
kubia-6b78f8d47-6vblm    0/2     CrashLoopBackOff   5
4m7s
```

There's your problem (or as you'll learn soon, your blessing)! The pod is shown as not ready, but I guess you've been expecting that, right? What has happened?

UNDERSTANDING HOW A READINESS PROBE PREVENTS BAD VERSIONS FROM BEING ROLLED OUT
As soon as your new pod starts, the readiness probe starts being hit every second (you set the probe's interval to one second in the pod spec). On the fifth request the readiness probe

began failing, because your app starts returning HTTP status code 500 from the fifth request onward.
As a result, the pod is removed as an endpoint from the service (see figure). By the time you start hitting the service in the curl loop, the pod has already been marked as not ready. This explains why you never hit the new pod with curl. And that's exactly what you want, because you don't want clients to hit a pod that's not functioning properly.



But what about the rollout process? The rollout status command shows only one new replica has started. Thankfully, the rollout process will not continue, because the new pod will never become available. To be considered available, it needs to be ready for at least 10 seconds. Until it's available, the rollout process will not create any new pods, and it also won't remove any original pods because you've set the maxUnavailable property to 0.

CONFIGURING A DEADLINE FOR THE ROLLOUT

By default, after the rollout can't make any progress in 10 minutes, it's considered as failed. If you use the kubectl describe deployment command, you'll see it display a ProgressDeadlineExceeded condition, as shown in the following listing.

```
$ kubectl describe deploy kubia
[...]
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
```

```
  Progressing    False   ProgressDeadlineExceeded
OldReplicaSets:  kubia-55d48448fd (3/3 replicas created)
NewReplicaSet:   kubia-6b78f8d47 (1/1 replicas created)
Events:
[...]
```

The time after which the Deployment is considered failed is configurable through the progressDeadlineSeconds property in the Deployment spec.

ABORTING A BAD ROLLOUT

Because the rollout will never continue, the only thing to do now is abort the rollout by undoing it:

```
$ kubectl rollout undo deployment kubia
deployment.apps/kubia rolled back
```

Declarative vs. Imperative Object Management

The Kubernetes kubectl command-line tool is used to create, update, and manage Kubernetes objects and supports imperative commands, imperative object configuration as well as declarative object configuration. Let's go through a real-world example that actually demonstrates the difference between an imperative/procedural configuration and a

declarative configuration in Kubernetes. First, let’s look at how kubectl can be used in an imperative manner.

In the example below, let’s create a script that will deploy an nginx service with three replicas, along with some annotations on the deployment.

```
#!/bin/sh
kubectl create deployment nginx-imperative --image=nginx:latest
#A
kubectl scale deployment/nginx-imperative --replicas 3
#B
kubectl annotate deployment/nginx-imperative environment=prod
#C
kubectl annotate deployment/nginx-imperative organization=sales
#D
```

#A Create a new deployment object called nginx-imperative.
#B Scale the nginx-imperative deployment to have three replicas of the pod.
#C Add an annotation with the key environment and value prod to the nginx-imperative deployment.
#D Add an annotation with the key organization and value sales to the nginx-imperative deployment.

Try running the script against your minikube cluster and check that the deployment was created successfully.

```
$ imperative-deployment.sh
deployment.apps/nginx-imperative created
deployment.apps/nginx-imperative scaled
deployment.apps/nginx-imperative annotated
deployment.apps/nginx-imperative annotated
```

```
$ kubectl get deployments
NAME             READY      UP-TO-DATE AVAILABLE  AGE
nginx-imperative 3/3        3          3          27s
```

Great! The deployment was created as expected. But now, let's edit our deployment.sh script to change the value of the organization annotation from sales to marketing and then rerun the script.

```
./imperative-deployment.sh
error: failed to create deployment: deployments.apps
"nginx-imperative" already exists
deployment.apps/nginx-imperative scaled
error: --overwrite is false but found the following declared
annotation(s): 'environment' already has a value (prod)
error: --overwrite is false but found the following declared
annotation(s): 'organization' already has a value (sales)
```

As you can see, the new script failed because the deployment and annotations already exist. To make it work, we would need to enhance our script with additional commands and logic to handle the update case as well as the creation case. Sure, this can be done, but it turns out we don't have to do all that work because kubectl already has the ability to examine the current state of the system and "do the right thing" using declarative object configuration.

The following manifest defines a deployment identical to the one created by our script (except the name of the deployment is nginx-declarative).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-declarative
  annotations:
    environment: prod
    organization: sales
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
```

We can use the semi-magical **kubectl apply** command to create the **nginx-declarative** deployment.

```
$ kubectl apply -f declarative-deployment.yaml
deployment.apps/nginx-declarative created
$ kubectl get deployment
NAME                 READY   UP-TO-DATE   AVAILABLE   AGE
nginx-declarative    3/3     3            3           15s
nginx-imperative     3/3     3            3           7m27s
```

The **kubectl apply** command will do a create operation if the **nginx-declarative** object does not exist or an update if it already exists.

Now, as with the imperative example, what if we want to change the value of the **organization** annotation from **sales** to **marketing**? Let's edit the **declarative-deployment.yaml** file. But before we run **kubectl apply** again, let's run **kubectl diff**.

```
[...]
-    organization: sales                          #A
+    organization: marketing
   creationTimestamp: "2020-11-29T15:59:39Z"
-  generation: 1
+  generation: 2                                    #B
[...]
```

#A The value of the organization label was changed from sales to marketing
#B The generation of this resource was changed by the system when doing kubectl apply

```
kubectl apply -f declarative-deployment2.yaml
deployment.apps/nginx-declarative configured
```

As you can see, **kubectl diff** properly identified that the organization was changed from **sales** to **marketing**. We also see that **kubectl apply** successfully applied the new changes.

In this exercise, both the imperative and declarative examples result in a deployment resource configured in exactly the same way. And at first glance, it may appear that the imperative approach is much simpler. It contains only a few lines of code compared to the verbosity of the declarative deployment spec, which is five times the size of the script. However, it contains problems that make it a poor choice to use in practice.

1. The code is not idempotent and may have different results if executed more than once. If executed a second time, an error will be thrown complaining that the deployment nginx already exists. In contrast, the deployment spec is idempotent,

meaning it can be applied as many times as needed, handling the case where the deployment already exists.

2. It is more difficult to manage changes to the resource over time, especially when the change is subtractive. Suppose you no longer wanted “organization” to be annotated on the deployment. Simply removing the “kubectl annotate” command from the scripted code would not help, since it would do nothing to remove the annotation from the existing deployment. A separate operation would be needed to remove it. On the other hand, with the declarative approach, you only need to remove the annotation line from the spec, and Kubernetes would take care of removing the annotation to reflect your desired state.
3. It is more difficult to understand changes. If a team member sent a pull request modifying the script to do something differently, it would be like any other source code review. The reviewer would need to mentally walk through the script’s logic to verify the algorithm achieves the desired outcome. There can even be bugs in the script. On the other hand, a pull request which changes a declarative deployment specification clearly shows the change to the desired state of the system. It is simpler to review, as there is no logic to review, only a configuration change.
4. The code is not atomic, meaning that if one of the four commands in the script failed, the state of the system would be partially changed and wouldn’t be in the original state, nor would it be in the desired state. With the declarative approach, the entire spec is received as a single request, and the system attempts to fulfill all aspects of the desired state as a whole.

As you can imagine, what started as a simple shell script, would need to become more and more complex to achieve idempotency. There are dozens of options available in the Kubernetes deployment spec, and with the scripted approach, if/else checks would need to be littered throughout the script, to understand existing state and conditionally modify the deployment.

How Declarative Configuration Work

As we saw in the previous exercise, declarative configuration management is powered by the **kubectl apply c**ommand. In contrast with imperative **kubectl** commands, like **scale** and **annotate**, the **kubectl apply** command has just one parameter: the path to the file containing the resource manifest.

kubectl apply -f ./resource.yaml

The command is responsible for figuring out which changes should be applied to the matching resource in the Kubernetes cluster and update the resource using the Kubernetes API. Let's learn more about the logic behind kubectl apply and get an understanding of what it can and cannot do. In order to understand which problems **kubectl apply** is solving, let's go through different scenarios using the Deployment resource we created above.
The simplest scenario is when the matching resource does not exist in the Kubernetes cluster. In this case, **kubectl** just creates a new resource using the manifest stored in the specified file.

If the matching resource exists, why doesn't **kubectl** just replace it? The answer is obvious if you take a look at the full manifest resource using the **kubectl get** command. Below is a partial listing of the Deployment resource that was created in the example above. Some parts of the manifest have been omitted for clarity (indicated with ellipses, e.g., "...").

```
$ kubectl get deployment nginx-declarative -o=yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
    environment: prod
    kubectl.kubernetes.io/last-applied-configuration: |
 [...]
    organization: marketing
  creationTimestamp: "2020-11-29T15:59:39Z"
  generation: 2
  name: nginx-declarative
  namespace: default
  resourceVersion: "8688"
  selfLink:
/apis/apps/v1/namespaces/default/deployments/nginx-declarative
  uid: 661a359e-0b25-4eb7-963b-63a9dc9ee282
spec:
  progressDeadlineSeconds: 600
  replicas: 3
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: nginx
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    …..
status:
   …...
```

As you may have noticed, the manifest of a live resource includes all the fields which were specified in the file plus dozens of new fields such as additional metadata, the **status** field as well as additional fields in the resource spec. All these additional fields are populated by the Deployment controller and contain important information about the running state of the resource. The controller populates information about resource state in the status field and applies default values of all unspecified optional fields, such as **revisionHistoryLimit** and strategy. In order to preserve this information, **kubectl apply** merges the manifest from the

specified file and the live resource manifest. As a result, the command updates only fields that are specified in the file, keeping everything else untouched. So if we decide to scale down the deployment and change the **replicas** field to 1 then **kubectl** changes only that field in the live resource and saves it back to Kubernetes using an update API.
In real life, we actually don't want to control all possible fields which influence resource behavior in a declarative way. It makes sense to leave some room for imperativeness and skip fields that should be changed dynamically. The **replicas** field of the Deployment resource is a perfect example. Instead of hard-coding the number of replicas you want to use, the Horizontal Pod Autoscaler can be used to dynamically scale up or scale down your application based on load.
Let's go ahead and remove the **replicas** field from the Deployment manifest. After applying this change, the replicas field is reset to the default value of one replica. But wait! The **kubectl apply** command updates only those fields which are specified in the file and ignores the rest. How does it know that the **replicas** field was deleted? The additional information which allows **kubectl** to handle the delete use case is hidden in an annotation of the live resource. Every time the **kubectl apply** command updates a resource, it saves the input manifest in the **kubectl.kubernetes.io/last-applied-configuration** annotation. So when the command is executed the next time, it retrieves the most recently applied manifest from the annotation, which represents the common ancestor of the new desired manifest and live resource manifest. This allows **kubectl** to execute a three-way diff/merge and properly handle the case where some fields are removed from the resource manifest.
Finally, let's discuss the situations where **kubectl apply** might not work as expected and should be used carefully.
First off, you typically should not mix imperative commands, such as **kubectl edit** or **kubectl scale,** with declarative resource management. This will make the current state not match the **last-applied-configuration** annotation and will defeat the merge algorithm kubectl uses to determine deleted fields. The typical scenario is when you experiment with the resource using kubectl edit and want to rollback changes by applying the original manifests stored in files. Unfortunately, it might not work since changes made by imperative kubectl commands are not stored anywhere. For example, if you temporarily add resource limits field to the deployment, the kubectl apply won't be able to remove it, since the limits field is not mentioned in the last-applied-configuration annotation or in the manifest from the file. So if you make any changes in an imperative way, you should be ready to undo the changes using imperative commands before continuing with declarative configuration.
You should also be careful when you want to stop managing fields declaratively. A common example of this problem is when adding the Horizontal Pod Autoscaler to manage scaling the number of replicas for an existing deployment. Typically, before introducing Horizontal Pod Autoscaler, the number of deployment replicas is managed declaratively. To pass control of the replicas field over to the Horizontal Pod Autoscaler, the replicas field must first be deleted from the file which contains the Deployment manifest. This is so the next **kubectl apply** does not override the replicas value set by the Horizontal Pod Autoscaler. However, don't forget that the replicas field might also be stored in the **last-applied-configuration** annotation. If that is the case, the missing replicas field in the manifest file will be treated as a field deletion, so whenever **kubectl apply** is run, the replicas value set imperatively by HPA will be removed from the live Deployment, and the Deployment will scale down to the default of one replica.

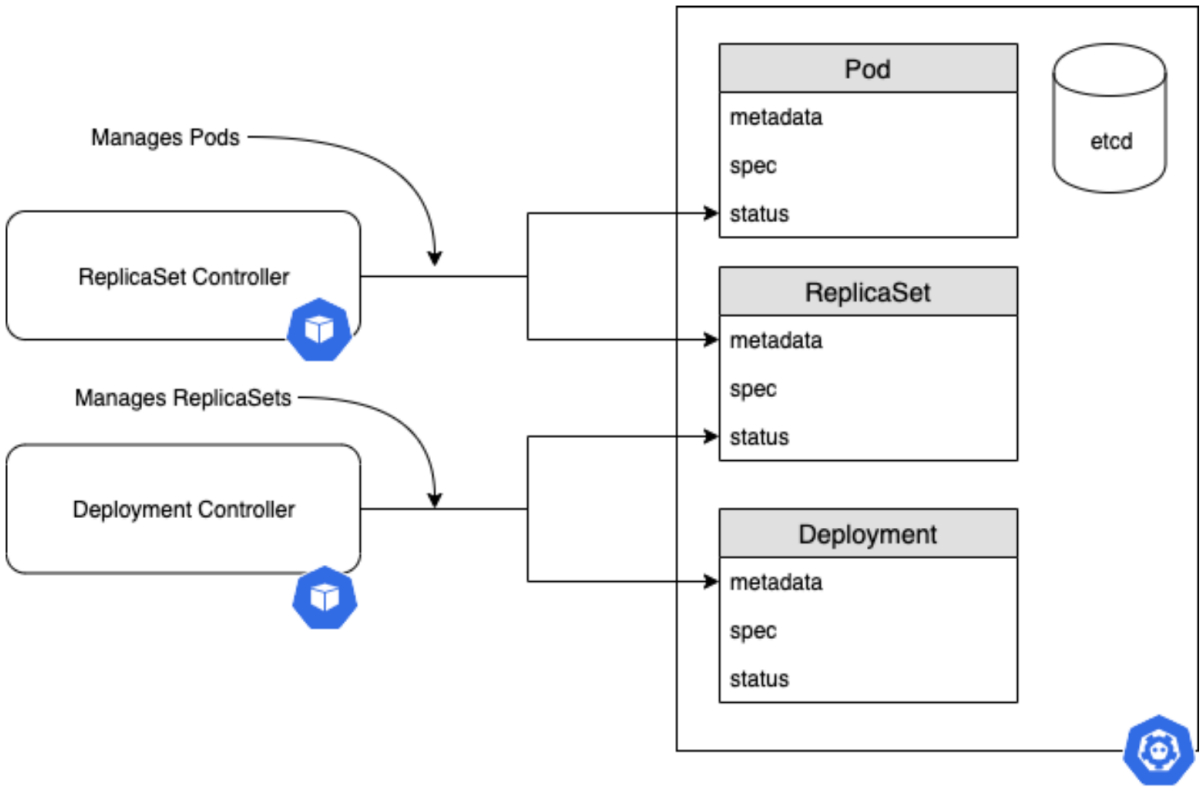
In this section, we covered the different mechanisms for managing Kubernetes objects: imperative and declarative. You also learned a little about the internals of kubectl and how it identifies changes to apply to live objects.

Controller Architecture

So far, we've learned about Kubernetes' declarative nature and the benefits it provides. Let's talk about what is behind each Kubernetes resource; the controller architecture. Understanding how controllers work will help us use Kubernetes more efficiently and understand how it can be extended.
Controllers are "brains" that understand what a particular **kind** of resource manifest means and execute the necessary work to make the actual state of the system match the desired state as described by the manifest. Each controller is typically responsible for only one resource type. Through listening to the API server events related to the resource type being managed, the controller continuously watches for changes to the resource's configuration and performs the necessary work to move the current state towards the desired state. An important feature of Kubernetes controllers is the ability to delegate work to other controllers. This layered architecture is very powerful and allows you to reuse functionality provided by different resource types effectively. Let's consider a concrete example to understand the delegation concept better.

Controller Delegation

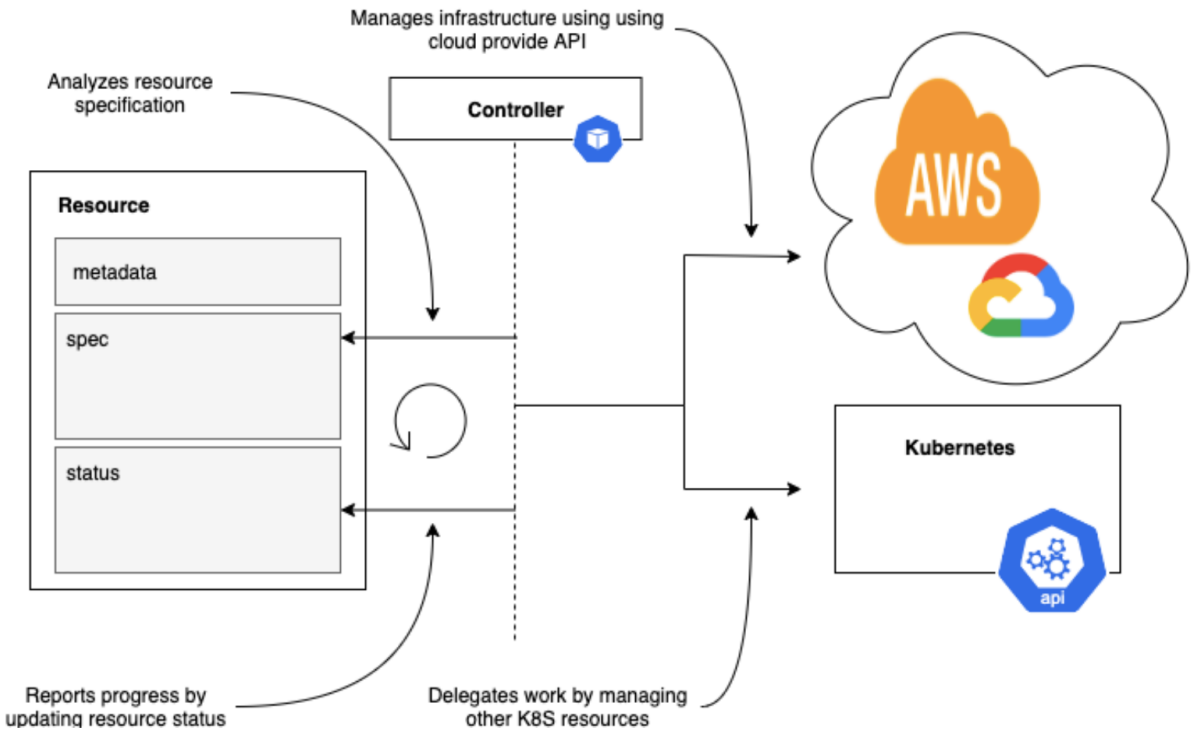
The Deployment, ReplicaSet, and Pod resources perfectly demonstrate how delegation empowers Kubernetes.



The Pod provides the ability to run one or more containers that have requested resources on a node in the cluster. This allows the Pod controller to focus simply on running an instance of an application and abstracts the logic related to infrastructure provisioning, scaling up and down, networking, and other complicated details, leaving those to other controllers. Although the Pod resource provides a lot of features, it is still not enough to run an application in production. We need to run multiple instances of the same application (for resiliency and performance), which means we need multiple Pods. The ReplicaSet controller solves this problem. Instead of directly managing multiple sets of containers, it orchestrates multiple Pods and delegates the container orchestration to the Pod resource. Similarly, the Deployment controller leverages functionality provided by ReplicaSets to implement various deployment strategies such as rolling update.
So, as you can see from this example, controller delegation allows Kubernetes to build progressively more complex resources from more simple ones.

Controller Pattern

Although all controllers have different responsibilities, the implementation of each controller follows the same simple pattern. Each controller runs an infinite loop, and every iteration reconciles the desired and the actual state of the cluster resources it is responsible for. During reconciliation, the controller is looking for differences between the actual and desired state and making the changes necessary to move the current state towards the desired state.
The desired state is represented by the spec field of the resource manifest. The question is, how does the controller know about the actual state? This information is available in the status field. After every successful reconciliation, the controller updates the status field. The status field provides information about cluster state to end-users and enables the work of higher-level controllers.



CONTROLLERS VS. OPERATORS

Two terms that are often confused are the term operator vs. controller. Operator is an application-specific controller that extends the Kubernetes API to create, configure, and manage instances of complex stateful applications on behalf of a Kubernetes user. It builds upon the basic Kubernetes resource and controller concepts, but also includes domain or application-specific knowledge to automate common tasks.

The terms operators and controllers are often confused since operator and controller are sometimes used interchangeably. However, another way to think about it is that the term “operator” is used to describe application-specific controllers. All operators use the controller pattern, but not all controllers are operators.

NGINX Operator

After learning about the controller fundamentals and the differences between Controllers and Operators, we are ready to implement an Operator! The sample operator will solve a real-life task: manage a suite of NGINX servers with pre-configured static content. The operator will allow the user to specify a list of NGINX servers and configure static files available on each server. The task is definitely not trivial and demonstrates the flexibility and power of Kubernetes.

DESIGN

As has been mentioned earlier in this chapter, Kubernetes’ architecture allows you to leverage the functionality of an existing controller through delegation effectively. Our NGINX controller is going to leverage Deployment resources to delegate the NGINX deployment task.

The next question is which resource should be used to configure the list of servers and customized static content. The most appropriate existing resource is ConfigMap. According to the official Kubernetes documentation, the ConfigMap is “an API object used to store non-confidential data in key-value pairs.” The ConfigMap can be consumed as environment variables, command-line arguments, or config files in a volume. The controller is going to create a Deployment for each ConfigMap and mount the ConfigMap data into the default NGINX static website directory.

IMPLEMENTATION

Once we've decided on the design of the main building blocks, it is time to write some code. Most Kubernetes related projects, including Kubernetes itself, are implemented using Golang. However, Kubernetes controllers can be implemented using any language, including Java, C++, or even Javascript. For the sake of simplicity, we are going to use a language that is most likely familiar to you: the **Bash** scripting language.

In the Controller Pattern section, we mentioned that each controller maintains an infinite loop and continuously reconciles the desired and actual state. In our example, the desired state is represented by the list of ConfigMaps. This most efficient way to loop through every ConfigMap change is through the use of the Kubernetes **watch** API. The **watch** feature is provided by the Kubernetes API for most resource types and allows the caller to get a notification when a resource is created, modified, or deleted. The **kubectl** utility allows watching for resource changes using the **get** command with the **--watch** flag. The **--output-watch-events** instructs **kubectl** to output the change type that might take one of the following values: **ADDED**, **MODIFIED**, or **DELETED**.

sample.yaml:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: sample
data:
  index.html: hello world
```

In one window, run the following command:

```
$ kubectl get --watch --output-watch-events configmap
```

In another terminal window, run **kubectl apply -f sample.yaml** to create the sample ConfigMap. Notice the new output in the window running the **kubectl --watch** command. Now run **kubectl delete -f sample.yaml.** You should now see a DELETED event appear.

```
$ kubectl get --watch --output-watch-events configmap
EVENT      NAME     DATA   AGE
ADDED      sample   1      13s
DELETED    sample   1      34s
```

After running this experiment manually, you should be able to see how we can write our NGINX operator as a bash script.
The **kubectl get --watch** command outputs a new line every time a ConfigMap resource is created, changed, or deleted. The script will consume the output of **kubectl get --watch** and either create a new Deployment or delete a Deployment depending on the output ConfigMap event type. Without further delay, the full operator implementation in the code listing below.

controller.sh

```
#!/usr/bin/env bash

kubectl get --watch --output-watch-events configmap \
-o=custom-columns=type:type,name:object.metadata.name \
--no-headers | \                                                    #A
while read next; do                                                 #B

    NAME=$(echo $next | cut -d' ' -f2)                              #C
    EVENT=$(echo $next | cut -d' ' -f1)

    case $EVENT in
        ADDED|MODIFIED)                                             #D
            kubectl apply -f - << EOF
apiVersion: apps/v1
kind: Deployment
metadata: { name: $NAME }
spec:
  selector:
    matchLabels: { app: $NAME }
  template:
    metadata:
      labels: { app: $NAME }
      annotations: { kubectl.kubernetes.io/restartedAt: $(date) }
    spec:
      containers:
      - image: nginx:1.7.9
        name: $NAME
        ports:
        - containerPort: 80
        volumeMounts:
        - { name: data, mountPath: /usr/share/nginx/html }
      volumes:
      - name: data
        configMap:
          name: $NAME
EOF
            ;;
        DELETED)                                                    #E
            kubectl delete deploy $NAME
            ;;
    esac
done
```

#A This kubectl command outputs all the events that occur for configmap objects
#B The output from kubectl is processed by this infinite loop
#C The name of the configmap and the event type is parsed from the kubectl output

#D If the configmap has been ADDED or MODIFIED, apply the NGINX deployment manifest (everything between the two EOF tags) for that configmap
#E If the configmap has been DELETED, delete the NGINX deployment for that configmap

TESTING

Now that the implementation is done, we are ready to test our controller. In real life, the controller is packaged into a Docker image and runs inside the cluster. For testing purposes, it is OK to run the controller outside of the cluster, which is exactly what we are going to do. Using instructions from Appendix A, start a minikube cluster, save the controller code into a file called controller.sh and start it using the bash command below.
Note: This example requires kubectl version 1.16 or later.

```
$ bash controller.sh
```

The controller is running and waiting for the ConfigMap. Let's create one.
We create the ConfigMap using the kubectl apply command.

```
$ kubectl apply -f sample.yaml
configmap/sample created
```

The controller notices the change and creates an instance of Deployment using the kubectl apply command.

```
$ bash controller.sh
deployment.apps/sample created
```