Table of Contents

[Introduction 4](#_Toc62323194)

[A Brief About Technologies # 4](#_Toc62323195)

[A Glimpse From The Past # 5](#_Toc62323196)

[A Short History of Infrastructure Management 6](#_Toc62323197)

[The Beginning # 6](#_Toc62323198)

[⚙️ Configuration Management # 6](#_Toc62323199)

[🛠 Configuration Management Tools # 7](#_Toc62323200)

[👍 Pros # 7](#_Toc62323201)

[👎 Cons # 7](#_Toc62323202)

[Virtual Machines # 7](#_Toc62323203)

[Mutability vs. Immutability # 8](#_Toc62323204)

[☁️ The Cloud Hosting # 9](#_Toc62323205)

[Modern Infrastructure # 10](#_Toc62323206)

[A Short History of Deployment Processes 10](#_Toc62323207)

[The Beginning # 11](#_Toc62323208)

[Configuration Management Tools # 11](#_Toc62323209)

[The Need of the Hour # 12](#_Toc62323210)

[Docker and Containers # 12](#_Toc62323211)

[Why Container Schedulers? # 12](#_Toc62323212)

[The Schedulers 13](#_Toc62323213)

[An Analogy # 13](#_Toc62323214)

[The Relatable Part # 14](#_Toc62323215)

[Why Use Schedulers? # 14](#_Toc62323216)

[The Container Schedulers # 15](#_Toc62323217)

[Why to Combine Containers and Schedulers? # 16](#_Toc62323218)

[What is Kubernetes? 16](#_Toc62323219)

[Don’t Run Containers Directly # 16](#_Toc62323220)

[The Kubernetes 17](#_Toc62323221)

[Why Kubernetes? # 17](#_Toc62323222)

[Concluding Remarks # 18](#_Toc62323223)

[PODS 18](#_Toc62323224)

[Getting Started with Pods 19](#_Toc62323225)

[Understanding Pods # 19](#_Toc62323226)

[Creating A Cluster # 19](#_Toc62323227)

[Defining Pods through Declarative Syntax 20](#_Toc62323228)

[Defining Pods Through Declarative Syntax # 20](#_Toc62323229)

[Looking into a Pod’s Definition # 21](#_Toc62323230)

[Components and Stages Involved in a Pod's Scheduling 24](#_Toc62323231)

[Components Involved in a Pod’s Scheduling # 24](#_Toc62323232)

[1. API Server # 24](#_Toc62323233)

[2. Scheduler # 25](#_Toc62323234)

[3. Kubelet # 25](#_Toc62323235)

[Sequential Breakdown of Events # 25](#_Toc62323236)

[Monitoring Health 26](#_Toc62323237)

[Why to Monitor Health? # 27](#_Toc62323238)

[Kubernetes Probes # 27](#_Toc62323239)

[Liveness Probe # 27](#_Toc62323240)

[Readiness Probe # 27](#_Toc62323241)

[Understanding the Updated Pod Definition # 28](#_Toc62323242)

[Liveness Probe in Action # 29](#_Toc62323243)

[Pods Are (Almost) Useless (By Themselves) # 30](#_Toc62323244)

[Getting Started with ReplicaSets 30](#_Toc62323245)

[Understanding ReplicaSets 30](#_Toc62323246)

[Creating A Cluster 31](#_Toc62323247)

[Creating ReplicaSets 32](#_Toc62323248)

[Looking into the Definition # 32](#_Toc62323249)

[Creating the ReplicaSet # 34](#_Toc62323250)

[Sequential Breakdown of the Process 36](#_Toc62323251)

[Sequential Breakdown of the Process # 36](#_Toc62323252)

[Operating ReplicaSets 39](#_Toc62323253)

[Deleting ReplicaSets # 39](#_Toc62323254)

[Re-using the Same Pods # 40](#_Toc62323255)

[Updating the Definition # 41](#_Toc62323256)

[Self-healing in Action # 42](#_Toc62323257)

[Destroying a Pod # 42](#_Toc62323258)

[Removing a label # 43](#_Toc62323259)

[Re-adding the Label # 44](#_Toc62323260)

[Operating ReplicaSets 45](#_Toc62323261)

[Deleting ReplicaSets # 45](#_Toc62323262)

[Re-using the Same Pods # 46](#_Toc62323263)

[Updating the Definition # 47](#_Toc62323264)

[Self-healing in Action # 48](#_Toc62323265)

[Destroying a Pod # 48](#_Toc62323266)

[Removing a label # 49](#_Toc62323267)

[Re-adding the Label # 50](#_Toc62323268)

[Getting Started with Volumes 51](#_Toc62323269)

[State Preservation # 51](#_Toc62323270)

[The Volumes # 51](#_Toc62323271)

[Copying the Files # 52](#_Toc62323272)

[Creating a Cluster # 52](#_Toc62323273)

[Accessing Host’s Resources through hostPath Volumes 53](#_Toc62323274)

[Building Docker Images # 53](#_Toc62323275)

[Creating a Pod with Docker Image # 54](#_Toc62323276)

[Creating a Pod with hostPath # 55](#_Toc62323277)

[The hostPath Volume # 56](#_Toc62323278)

[Types of Mounts in hostPath # 57](#_Toc62323279)

[Running the Pod after mounting hostPath 57](#_Toc62323280)

[Creating and Testing the Pod # 57](#_Toc62323281)

[Diagram

Description automatically generated 58](#_Toc62323282)

[Playing Around with Docker # 59](#_Toc62323283)

[Destroying the Pod # 61](#_Toc62323284)

**Introduction**

In this lesson, we will discuss briefly about the currently used technologies and will peek a bit into the past.

* [A Brief About Technologies](#A Brief About Technologies )
* [A Glimpse From The Past](#A Glimpse From The Past )

## A Brief About Technologies [#](https://www.educative.io/courses/practical-guide-to-kubernetes/R8rn5n66WVK#a-brief-about-technologies)

Technology is changing so fast that it is very hard, if not impossible to follow. The moment we learn about a new technology, it is already obsolete and replaced with something else.

Take **containers** as an example. Docker appeared only a few years ago, and everyone is already using it for a myriad of scenarios. Still, even though it is a very young product, it changed many times over. Just when we learned how to use docker run, we were told that it is obsolete and should be replaced with docker-compose up. We started converting all our docker run commands into Compose YAML format.

The moment we finished the conversion, we learned that containers should not be run directly. We should use a container scheduler instead. To make things more complicated, we had to make a selection between **Mesos and Marathon, Docker Swarm, or Kubernetes**.

We can choose to ignore the trends but that would mean that we would fall behind the competition. There is no alternative to a constant struggle to be competitive. Once we drop our guard and stop learning and improving, the competition will take over our business. Everyone is under pressure to improve, even highly regulated industries. Innovation is impossible until we manage to get to the present tense. Only once we master what others are doing today, we can move forward and come up with something new.

Today, container schedulers are a norm. They are not the thing of the future. They are the present. They are here to stay, even though it is likely that they will change a lot in the coming months and years. Understanding container schedulers is extremely important.

**Kubernetes** is the most widely used container scheduler that has a massive community behind it.

Before we dive into Kubernetes, it might be worthwhile going through some history in an attempt to understand some of the problems we were trying to solve, as well as some of the challenges we were facing.

## A Glimpse From The Past [#](https://www.educative.io/courses/practical-guide-to-kubernetes/R8rn5n66WVK#a-glimpse-from-the-past)

Picture a young boy. He just finished a few months worth of work. He’s proud of what he accomplished yet fearful whether it will work. He did not yet try it out on a “real” server. This will be the first time he’ll deliver the fruits of his work.

He takes a floppy disk out from a drawer, inserts it into his computer, and copies the files he compiled previously. He feels fortunate that perforated cards are a thing of the past.

He gets up from his desk, exits the office, and walks towards his car. It will take him over two hours to get to the building with servers. He’s not happy with the prospect of having to drive for two hours, but there is no better alternative. He could have sent the floppy with a messenger, but that would do no good since he wants to install the software himself. He needs to be there. There is no remote option.

A while later, he enters the room with the servers, inserts the floppy disk, and copies and installs the software. Fifteen minutes later, his face shows signs of stress. Something is not working as expected. There is an unforeseen problem. He’s collecting outputs and writing notes. He’s doing his best to stay calm and gather as much info as he can. He’s dreading a long ride back to his computer and days, maybe even weeks, until he figures out what caused the problem and fixes it. He’ll be back and install the fix. Perhaps it will work the second time. More likely it won’t.

So, this was a glimpse from the past. We can imagine the uncertainty and the effort one needed to put for getting a simple deployment task done.

# A Short History of Infrastructure Management

In this lesson, we will peek into the past and go through a short history of infrastructure management.

* [The Beginning](#The Beginning )
* [⚙️ Configuration Management](#⚙️ Configuration Management )
  + [🛠 Configuration Management Tools](#🛠 Configuration Management Tools )
  + [👍 Pros](#👍 Pros )
  + [👎 Cons](#👎 Cons )
* [Virtual Machines](#Virtual Machines )
  + [Mutability vs. Immutability](#Mutability vs. Immutability )
* [☁️ The Cloud Hosting](#☁️ The Cloud Hosting )
* [Modern Infrastructure](#Modern Infrastructure )

## The Beginning [#](https://www.educative.io/courses/practical-guide-to-kubernetes/m745rL5WMBO#the-beginning)

A long time ago in a galaxy far, far away…

We would order servers and wait for months until they arrive. To make our misery worse, even after they come, we’d wait for weeks, sometimes even months, until they are placed in racks and provisioned.

At that time, only a select few people could access these servers. If someone does something that should not be done, we could face an extended downtime. On top of that, nobody knew what was running on those servers.

Manual provisioning and installations were a nightmare because even after putting a lot of effort into documentation, given enough time, the state of the servers would always diverge from the documentation. **Sysadmins** were the key people without whom no one can handle these servers.

## ⚙️ Configuration Management [#](https://www.educative.io/courses/practical-guide-to-kubernetes/m745rL5WMBO#%EF%B8%8F-configuration-management)

To manage the configuration means to track and control changes in the software. Configuration management tools enable us to determine what was changed, who changed it and much more.

### 🛠 Configuration Management Tools [#](https://www.educative.io/courses/practical-guide-to-kubernetes/m745rL5WMBO#configuration-management-tools)

Then came configuration management tools. We got **CFEngine**.

### 👍 Pros [#](https://www.educative.io/courses/practical-guide-to-kubernetes/m745rL5WMBO#pros)

It was based on promise theory and was capable of putting a server into the desired state no matter what its actual state was.

It allowed us to specify the state of static infrastructure and have a reasonable guarantee that it will be achieved.

Another big advantage it provided is the ability to have, more or less, the same setup for different environments. Servers dedicated to testing could be (almost) the same as those assigned to production.

### 👎 Cons [#](https://www.educative.io/courses/practical-guide-to-kubernetes/m745rL5WMBO#cons)

Unfortunately, usage of CFEngine and similar tools were not yet widespread. We had to wait for virtual machines before automated configuration management became a norm. However, CFEngine was not designed for virtual machines. They were meant to work with static, bare metal servers. Still, CFEngine was a massive contribution to the industry even though it failed to get widespread adoption.

After CFEngine came Chef, Puppet, Ansible, Salt, and other similar tools. We’ll go back to these tools soon. For now, let’s turn to the next evolutionary improvement.

Besides forcing us to be patient, physical servers were a massive waste in resource utilization. They came in predefined sizes and, since waiting time was considerable, we often opted for big ones. The bigger, the better. That meant that an application or a service usually required less CPU and memory than the server offered. Unless you do not care about costs, that meant that we’d deploy multiple applications to a single server. The result was a dependencies nightmare. We had to choose between freedom and standardization.

Freedom meant that different applications could use different runtime dependencies while standardization involves systems architects deciding the only right way to develop and deploy something.

## Virtual Machines [#](https://www.educative.io/courses/practical-guide-to-kubernetes/m745rL5WMBO#virtual-machines)

Then came Virtual machines and broke everyone’s happiness.

Virtual machines (VMs) were a massive improvement over bare metal infrastructure.

* They allowed us to be more precise with hardware requirements.
* They could be created and destroyed quickly.
* They could differ i.e. a single physical server could have multiple VMs running in isolation. One VM could host a Java application, and the other could be dedicated to Ruby on Rails.
* We could get them in a matter of minutes, instead of waiting for months. Still, it took quite a while until “could” became “can”.

Even though the advantages brought by VMs were numerous, years passed until they were widely adopted. Even then, the adoption was usually wrong. Companies often moved the same practices used with bare metal servers into virtual machines. We could have identical servers in different environments. Companies started copying VMs. While that was much better than before, it did not solve the problem of missing documentation and the ability to create VMs from scratch. Still, multiple identical environments are better than one, even if that meant that we don’t know what’s inside.

### Mutability vs. Immutability [#](https://www.educative.io/courses/practical-guide-to-kubernetes/m745rL5WMBO#mutability-vs-immutability)

The configuration management tools helped spread the adoption of “infrastructure as code” principles. But the problem was they were designed with static infrastructure in mind. On the other hand, VMs opened the doors to dynamic infrastructure where VMs are continuously created and destroyed. Mutability and constant creation and destruction were clashing. Mutable infrastructure is well suited for static infrastructure. It does not respond well to challenges brought with dynamic nature of modern data centers. Mutability (changeable at runtime) had to give way to immutability (nothing can be tweaked at runtime).

When ideas behind immutable infrastructure started getting traction, people began combining them with the concepts behind configuration management. However, tools available at that time were not fit for the job. They (Chef, Puppet, Ansible, and the like) were designed with the idea that servers are brought into the desired state at runtime. Immutable processes, on the other hand, assume that (almost) nothing is changeable at runtime. Artifacts were supposed to be created as immutable images. In case of infrastructure, that meant that VMs are created from images, and not changed at runtime. If an upgrade is needed, a new image should be created followed with a replacement of old VMs with new ones based on the new image. Such processes brought speed and reliability. With proper tests in place, immutable is always more reliable than mutable.

Subsequently, we got tools capable of building VM images. Today, they are ruled by Packer. Configuration management tools quickly jumped on board, and their vendors told us that they work equally well for configuring images as servers at runtime. However, that was not the case due to the logic behind those tools. They are designed to put a server that is in an unknown state into the desired state. They assume that we are not sure what the current state is. VM images, on the other hand, are always based on an image with a known state. If for example, we choose Ubuntu as a base image, we know what’s inside it.

Adding additional packages and configurations is easy. There is no need for things like “if this then that, otherwise something else.” A simple shell script is as good as any configuration management tool when the current state is known. Creating a VM image is reasonably straightforward with Packer alone. Still, not all was lost for configuration management tools. We could still use them to orchestrate the creation of VMs based on images and, potentially, do some runtime configuration that couldn’t be baked in. Right?

## ☁️ The Cloud Hosting [#](https://www.educative.io/courses/practical-guide-to-kubernetes/m745rL5WMBO#%EF%B8%8F-the-cloud-hosting)

The way we orchestrate infrastructure had to change as well. A higher level of dynamism and elasticity was required. That became especially evident with the emergence of cloud hosting providers like Amazon Web Services (AWS) and, later on, Azure and GCE.

They showed us what can be done. While some companies embraced the cloud, others went into defensive positions. “We can build an internal cloud”, “AWS is too expensive”, “I would, but I can’t because of legislation”, and “our market is different”, are only a few ill-conceived excuses often given by people who are desperately trying to maintain status quo. That is not to say that there is no truth in those statements but that, more often than not, they are used as an excuse, not for real reasons.

Still, the cloud did manage to become the way to do things, and companies moved their infrastructure to one of the providers. Or, at least, started thinking about it. The number of companies that are abandoning on-premise infrastructure is continuously increasing, and we can safely predict that the trend will continue.

Still, the question remains. How do we manage infrastructure in the cloud with all the benefits it gives us? How do we handle its highly dynamic nature? The answer came in the form of vendor-specific tools like CloudFormation or agnostic solutions like Terraform. When combined with tools that allow us to create images, they represent a new generation of configuration management. We are talking about full automation backed by immutability.

## Modern Infrastructure [#](https://www.educative.io/courses/practical-guide-to-kubernetes/m745rL5WMBO#modern-infrastructure)

We’re living in an era without the need to SSH into servers.

Today, modern infrastructure is created from immutable images. Any upgrade is performed by building new images and performing rolling updates that will replace VMs one by one. Infrastructure dependencies are never changed at runtime. Tools like Packer, Terraform, CloudFormation, and the like are the answer to today’s problems.

One of the inherent benefits behind immutability is a clear division between infrastructure and deployments. Until not long ago, the two meshed together into an inseparable process. With infrastructure becoming a service, deployment processes can be clearly separated, thus allowing different teams, individuals, and expertise to take control.

We’ll need to go back in time one more time and discuss the history of deployments. Did they change as much as infrastructure?

# A Short History of Deployment Processes

In this lesson, we will peek into the past and go through a short history of deployment processes.

* [The Beginning](#_The_Beginning_)
* [Configuration Management Tools](#Configuration Management Tools )
* [The Need of the Hour](#The Need of the Hour )
* [Docker and Containers](#The Need of the Hour )
* [Why Container Schedulers?](#Why Container Schedulers? )

## The Beginning [#](https://www.educative.io/courses/practical-guide-to-kubernetes/R8Q72BJVOmq#the-beginning)

In the beginning, there were no package managers. There were no JAR, WAR, RPM, DEB, and other package formats. Package managers typically maintain a database of software dependencies and version information to prevent software mismatches and missing prerequisites. The best we could do at that time was to zip files that form a release. More likely, we’d manually copy files from one place to another. When this practice is combined with bare-metal servers which were intended to last forever, the result was living hell. After some time, no one knew what was installed on the servers. Constant overwrites, reconfigurations, package installations, and mutable types of actions resulted in unstable, unreliable, and undocumented software running on top of countless OS patches.

## Configuration Management Tools [#](https://www.educative.io/courses/practical-guide-to-kubernetes/R8Q72BJVOmq#configuration-management-tools)

The emergence of configuration management tools (e.g., CFEngine, Chef, Puppet, and so on) helped to decrease the mess. Still, they improved OS setups and maintenance, more than deployments of new releases. They were never designed to do that even though the companies behind them quickly realized that it would be financially beneficial to extend their scope.

Even with configuration management tools, the problems with having multiple services running on the same server persisted. Different services might have different needs, and sometimes those needs clash. One might need JDK6 and the other JDK7. A new release of the first one might require JDK to be upgraded to a new version, but that might affect some other service on the same server. Conflicts and operational complexity were so common that many companies would choose to standardize. As we discussed, standardization is an innovation killer. The more we standardize, the less room there is for coming up with better solutions. Even if that’s not a problem, standardization with clear isolation means that it is very complicated to upgrade something. Effects could be unforeseen and the sheer work involved to upgrade everything at once is so significant that many choose not to upgrade for a long time (if ever). Many end up stuck with old stacks for a long time.

## The Need of the Hour [#](https://www.educative.io/courses/practical-guide-to-kubernetes/R8Q72BJVOmq#the-need-of-the-hour)

We needed process isolation that does not require a separate VM for each service. At the same time, we had to come up with an immutable way to deploy software. Mutability was distracting us from our goal to have reliable environments. With the emergence of virtual machines, immutability became feasible. Instead of deploying releases by doing updates at runtime, we could create new VMs with not only OS and patches but also our own software baked in. Each time we wanted to release something, we could create a new image, and instantiate as many VMs as we need. We could do immutable rolling updates. Still, not many of us did that. It was too expensive, both regarding resources as well as time. The process was too long. Even if that would not matter, having a separate VM for each service would result in too much unused CPU and memory.

## Docker and Containers [#](https://www.educative.io/courses/practical-guide-to-kubernetes/R8Q72BJVOmq#docker-and-containers)

Fortunately, Linux got namespaces, cgroups, and other things that are together known as containers. They were lightweight, fast, and cheap. They provided process isolation and quite a few other benefits. Unfortunately, they were not easy to use. Even though they’ve been around for a while, only a handful of companies had the know-how required for their beneficial utilization. We had to wait for Docker to emerge to make containers easy to use and thus accessible to all.

Today, **containers** are the preferable way to package and deploy services. They are the answer to immutability we were so desperately trying to implement. They provide necessary isolation of processes, optimized resource utilization, and quite a few other benefits. And yet, we already realized that we need much more.

## Why Container Schedulers? [#](https://www.educative.io/courses/practical-guide-to-kubernetes/R8Q72BJVOmq#why-container-schedulers)

It’s not enough to run containers. We need to be able to scale them, to make them fault tolerant, to provide transparent communication across a cluster, and many other things. Containers are only a low-level piece of the puzzle. The real benefits are obtained with tools that sit on top of containers. Those tools are today known as container schedulers. They are our interface. We do not manage containers, they do.

In case you are not already using one of the container schedulers, you might be wondering what they are.

# The Schedulers

In this lesson, we will get familiar with the container schedulers.

* [An Analogy](#An Analogy )
* The Relatable Part
* Why Use Schedulers?
* The Container Schedulers
* Why to Combine Containers and Schedulers?

## An Analogy [#](https://www.educative.io/courses/practical-guide-to-kubernetes/3wAm2lrQJpA#an-analogy)

Picture some young teenagers. After school, they’d go to a courtyard and play soccer. That was an exciting sight. A random number of kids running around the yard without any orchestration. There was no offense and no defense. They’d just run after a ball.

Everyone moves forward towards the ball, someone kicks it to the left, and kids move in that direction, only to start running back because someone kicked the ball again. The strategy was simple. Run towards the ball, kick it if you can, wherever you can, repeat. It’s hard to understand how anyone managed to score. It was complete randomness applied to a bunch of kids. There was no strategy, no plan, and no understanding that winning required coordination.

If that was a “real” team, they’d need a coach. They’d need someone to tell us what the strategy is, who should do what, and when to go on the offense or fall back to defend the goal. They’d need someone to orchestrate them.

The field (a cluster) had a random number of people (services) with the common goal (to win). Since anyone could join the game at any time, the number of people (services) was continually changing. Someone would be injured and would have to be replaced or, when there was no replacement, the rest of us would have to take over his tasks (self-healing).

The following illustrations will give you a basic idea of a node and a cluster.

**1** of 2

## The Relatable Part [#](https://www.educative.io/courses/practical-guide-to-kubernetes/3wAm2lrQJpA#the-relatable-part)

Those football games can be easily translated into clusters. Just as the kids needed someone to tell them what to do (a coach), clusters need something to orchestrate all the services and resources. Both need not only to make up-front decisions, but also to continuously watch the game/cluster, and adapt the strategy/scheduling depending on the internal and external influences. Kids needed a coach and clusters need a scheduler. They need a framework that will decide where a service should be deployed and make sure that it maintains the desired run-time specification.

## Why Use Schedulers? [#](https://www.educative.io/courses/practical-guide-to-kubernetes/3wAm2lrQJpA#why-use-schedulers)

A cluster scheduler has quite a few goals.

* It makes sure that resources are used efficiently and within constraints.
* It makes sure that services are (almost) always running.
* It provides fault tolerance and high availability.
* It makes sure that the specified number of replicas are deployed.
* It makes sure that the desired state requirement of a service or a node is (almost) always fulfilled. Instead of using imperative methods to achieve our goals, with schedulers, we can be declarative.
* We can tell a scheduler what the desired state is, and it will do its best to ensure that our desire is (almost) always fulfilled. For example, instead of executing a deployment process five times hoping that we’ll have five replicas of a service, we can tell a scheduler that our desired state is to have the service running with five replicas.

The difference between imperative and declarative methods might seem subtle but, in fact, is enormous. With a declarative expression of the desired state, a scheduler can monitor a cluster and perform actions whenever the actual state does not match the desired. Compare that to an execution of a deployment script. Both will deploy a service and produce the same initial result. However, the script will not make sure that the result is maintained over time. If an hour later, one of the replicas fail, our system will be compromised.

Traditionally, we were solving that problem with a combination of alerts and manual interventions. An operator would receive a notification that a replica failed, he’d log in to the server, and restart the process. If the whole server is down, the operator might choose to create a new one, or he might deploy the failed replica to one of the other servers. But, before doing that, he’d need to check which server has enough available memory and CPU. All that, and much more, is done by schedulers without human intervention.

Think of schedulers as operators who are continually monitoring the system and fixing discrepancies between the desired and the actual state. The difference is that schedulers are infinitely faster and more accurate. They do not get tired, they do not need to go to the bathroom, and they do not require paychecks. They are machines or, to be more precise, software running on top of them.

## The Container Schedulers [#](https://www.educative.io/courses/practical-guide-to-kubernetes/3wAm2lrQJpA#the-container-schedulers)

That leads us to container schedulers. How do they differ from schedulers in general?

Container schedulers are based on the same principles as schedulers in general. The significant differences between a scheduler and a container scheduler are:

* They are using containers as the deployment units.
* They are deploying services packaged as container images.
* They are trying to collocate them depending on desired memory and CPU specifications.
* They are making sure that the desired number of replicas are (almost) always running.

All in all, they do what other schedulers do but with containers as the lowest and the only packaging unit. And that gives them a distinct advantage. They do not care what’s inside. From a scheduler’s point of view, all containers are the same.

## Why to Combine Containers and Schedulers? [#](https://www.educative.io/courses/practical-guide-to-kubernetes/3wAm2lrQJpA#why-to-combine-containers-and-schedulers)

Containers provide benefits that other deployment mechanisms do not.

* Services deployed as containers are isolated and immutable.
* Isolation provides reliability.
* Isolation helps with networking and volume management. It avoids conflicts. It allows us to deploy anything, anywhere, without worrying whether that something will clash with other processes running on the same server.
* Schedulers, combined with containers and virtual machines, provide the ultimate cluster management nirvana.
* They allow us to combine the developer’s necessity for rapid and frequent deployments with a sysadmin’s goals of stability and reproducibility.

And all that leads us to Kubernetes…

# What is Kubernetes?

This lesson will get us introduced to the world of Kubernetes.

* [Don’t Run Containers Directly](https://www.educative.io/courses/practical-guide-to-kubernetes/NEx4zlXE4P2#dont-run-containers-directly)
* [Why Kubernetes?](https://www.educative.io/courses/practical-guide-to-kubernetes/NEx4zlXE4P2#why-kubernetes)
* [Concluding Remarks](https://www.educative.io/courses/practical-guide-to-kubernetes/NEx4zlXE4P2#concluding-remarks)

## Don’t Run Containers Directly [#](https://www.educative.io/courses/practical-guide-to-kubernetes/NEx4zlXE4P2#dont-run-containers-directly)

To understand Kubernetes, it is important to realize that running containers directly is a bad option for most use cases. Containers are low-level entities that require a framework on top. They need something that will provide all the additional features we expect from services deployed to clusters. In other words, containers are handy but are not supposed to be run directly.

The reason is simple. Containers, by themselves, do not provide fault tolerance. They cannot be deployed easily to the optimum spot in a cluster, and, to cut a long story short, are not operator friendly. That does not mean that containers by themselves are not useful. They are, but they require much more if we are to harness their real power. If we need to operate containers at scale, be fault tolerant and self-healing, and have the other features we expect from modern clusters, we need more. We need at least a scheduler, probably more.

## The Kubernetes

Kubernetes was first developed by a team at Google. It is based on their experience from running containers at scale for years. Later on, it was donated to [Cloud Native Computing Foundation (CNCF)](https://www.cncf.io/). It is a true open source project with probably the highest velocity in history.

### Why Kubernetes? [#](https://www.educative.io/courses/practical-guide-to-kubernetes/NEx4zlXE4P2#why-kubernetes)

Let’s discuss how Kubernetes is not only a container scheduler but a lot more.

* We can use it to deploy our services, to roll out new releases without downtime, and to scale (or de-scale) those services.
* It is portable.
* It can run on a public or private cloud.
* It can run on-premise or in a hybrid environment.
* We can move a Kubernetes cluster from one hosting vendor to another without changing (almost) any of the deployment and management processes.
* Kubernetes can be easily extended to serve nearly any needs. We can choose which modules we’ll use, and we can develop additional features ourselves and plug them in.
* Kubernetes will decide where to run something and how to maintain the state we specify.
* Kubernetes can place replicas of a service on the most appropriate server, restart them when needed, replicate them, and scale them.
* Self-healing is a feature included in its design from the start. On the other hand, self-adaptation is coming soon as well.
* Zero-downtime deployments, fault tolerance, high availability, scaling, scheduling, and self-healing add significant value in Kubernetes.
* We can use it to mount volumes for stateful applications.
* It allows us to store confidential information as secrets.
* We can use it to validate the health of our services.
* It can load balance requests and monitor resources.
* It provides service discovery and easy access to logs.

And so on and so forth.

## Concluding Remarks [#](https://www.educative.io/courses/practical-guide-to-kubernetes/NEx4zlXE4P2#concluding-remarks)

The list of what Kubernetes does is long and rapidly increasing. Together with Docker, it is becoming a platform that envelops the whole software development and deployment lifecycle.

The Kubernetes project has just started. It is in its infancy, and we can expect vast improvements and new features coming soon. Still, do not be fooled with “infancy”. Even though the project is young, it has one of the biggest communities behind it and is used in some of the biggest clusters in the world.

**Do not wait. Adopt it now!**

# PODS

**Getting Started with Pods**

In this lesson, we will understand Pods and create a Kubernetes cluster.

* [Understanding Pods](#Understanding Pods )
* [Creating A Cluster](#Creating A Cluster )

## Understanding Pods [#](https://www.educative.io/courses/practical-guide-to-kubernetes/7nzzrPvBqM1#understanding-pods)

**Pods** are equivalent to bricks we use to build houses. Both are uneventful and not much by themselves. Yet, they are fundamental building blocks without which we could not construct the solution we are set to build.

If you have used Docker or Docker Swarm, you’re probably used to thinking that a container is the smallest unit and that more complex patterns are built on top of it. With Kubernetes, the smallest unit is a Pod.

A **Pod** is a way to represent a running process in a cluster.

From the Kubernetes’ perspective, there’s nothing smaller than a Pod.

A Pod encapsulates one or more containers. It provides a unique network IP, attaches storage resources, and also decides how containers should run. Everything in a Pod is tightly coupled.

We should clarify that containers in a Pod are not necessarily made by Docker. Other container runtimes are supported as well. Still, at the time of this writing, Docker is the most commonly used container runtime, and all our examples will use it.

Since we cannot create Pods without a Kubernetes cluster, our first order of business is to create one.

ℹ️ All the commands from this chapter are available in the [03-pods.sh](https://gist.github.com/vfarcic/d860631d0dd3158c32740e9260c7add0) Gist.

## Creating A Cluster [#](https://www.educative.io/courses/practical-guide-to-kubernetes/7nzzrPvBqM1#creating-a-cluster)

We’ll create a local Kubernetes cluster using Minikube.

1

2

minikube start --vm-driver=virtualbox

kubectl get nodes

The **output** of the latter command is as follows.

1

2

NAME     STATUS ROLES  AGE VERSION

minikube Ready  master 47s v1.14.0

To simplify the process and save you from writing all the configuration files, we’ll clone the GitHub [repository](https://github.com/vfarcic/k8s-specs). It contains everything we’ll need for this chapter, as well as for most of the others in this course.

1

2

git clone https://github.com/vfarcic/k8s-specs.git

cd k8s-specs

We cloned the repository and entered into the directory that was created.

**Defining Pods through Declarative Syntax**

In this lesson, we will create and run Pods using declarative syntax.

**We'll cover the following**

* [Defining Pods Through Declarative Syntax](https://www.educative.io/courses/practical-guide-to-kubernetes/gknBZkWBDwl#defining-pods-through-declarative-syntax)
* [Looking into a Pod’s Definition](https://www.educative.io/courses/practical-guide-to-kubernetes/gknBZkWBDwl#looking-into-a-pods-definition)

**Defining Pods Through Declarative Syntax**[#](https://www.educative.io/courses/practical-guide-to-kubernetes/gknBZkWBDwl#defining-pods-through-declarative-syntax)

Even though a Pod can contain any number of containers, the most common use case is to use the **single-container-in-a-Pod** model. In such a case, a Pod is a *wrapper* around one container. From Kubernetes’ perspective, a Pod is the smallest unit.

We **cannot** tell Kubernetes to run a container. Instead, we ask it to create a Pod that wraps around a container.

**Looking into a Pod’s Definition**[#](https://www.educative.io/courses/practical-guide-to-kubernetes/gknBZkWBDwl#looking-into-a-pods-definition)

Let’s take a look at a simple Pod definition by accessing the db.yml file from the cloned git [repository](https://github.com/vfarcic/k8s-specs/blob/master/pod/db.yml).

1

cat pod/db.yml

The **output** is as follows.

Text

Description automatically generated

Let’s analyze the various sections in the output definition of a Pod.

* **Line 1-2:** We’re using v1 of Kubernetes Pods API. Both apiVersion and kind are mandatory. That way, Kubernetes knows what we want to do (create a Pod) and which API version to use.
* **Line 3-7:** The next section is metadata. It provides information that does not influence how the Pod behaves. We used metadata to define the name of the Pod (db) and a few labels. Later on, when we move into *Controllers*, labels will have a practical purpose. For now, they are purely informational.
* **Line 8:** The last section is the spec in which we defined a single container. As you might have guessed, we can have multiple containers defined as a Pod. Otherwise, the section would be written in singular (container without s). We’ll explore multi-container Pods later.
* **Line 12:** In our case, the container is defined with the name (db), the image (mongo), the command that should be executed when the container starts (mongod)
* **Line 13:** Finally, the set of arguments. The arguments are defined as an array with, in this case, two elements (--rest and --httpinterface).

We won’t go into details of everything you can use to define a Pod. Throughout the course, you’ll see quite a few other commonly (and not so commonly) used things we should define in Pods. Later on, when you decide to learn all the possible arguments you can apply, explore the official, and ever-changing, [Pod v1 core](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.18/#pod-v1-core) documentation.

Let’s create the Pod defined in the db.yml file.

1

kubectl create -f pod/db.yml

You’ll notice that we did not need to specify pod in the command. The command will create the kind of resource defined in the pod/db.yml file. Later on, you’ll see that a single YAML file can contain definitions of multiple resources.

Let’s take a look at the Pods in the cluster.

1

kubectl get pods

The **output** is as follows.

NAME READY STATUS RESTARTS AGE

db 1/1 Running 0 11s

Our Pod named db is **up and running!**

In some cases, you might want to retrieve a bit more information by specifying wide output.

1

kubectl get pods -o wide

The **output** is as follows.

1

2

NAME READY STATUS  RESTARTS AGE IP         NODE     NOMINATED NODE  READINESS GATES

db   1/1   Running 0        1m  172.17.0.4 minikube <none>          <none>

As you can see, we got two additional columns; the IP and the NODE.

If you’d like to parse the output, using json format is probably the best option.

1

kubectl get pods -o json

The output is too big to be presented here, especially since we won’t go through all the information provided through the json output format.

When we want more information than provided with the default output, but still in a format that is human-friendly, yaml output is probably the best choice.

1

kubectl get pods -o yaml

Just as with the json output, we won’t go into details of everything we got from Kubernetes. With time, you’ll become familiar with all the information related to Pods. For now, we want to focus on the most important aspects.

Let’s introduce a new kubectl sub-command.

1

kubectl describe pod db

The describe sub-command returned details of the specified resource. In this case, the resource is the Pod named db.

The output is too big for us to go into every detail. Besides, most of it should be self-explanatory if you’re familiar with containers. Instead, we’ll briefly comment on the last section called events.

Text

Description automatically generated with medium confidence

The above output may appear a bit different and take some time to show up in its full form.

We can see that Pod was created. Even though the process was simple from a user’s perspective, quite a few things happened in the background.

**Components and Stages Involved in a Pod's Scheduling**

In this lesson, we will go through the stages involved in a Pod's creation.

* [Components Involved in a Pod’s Scheduling](https://www.educative.io/courses/practical-guide-to-kubernetes/myPLNMjMln3#components-involved-in-a-pods-scheduling)
  + [1. API Server](https://www.educative.io/courses/practical-guide-to-kubernetes/myPLNMjMln3#1-api-server)
  + [2. Scheduler](https://www.educative.io/courses/practical-guide-to-kubernetes/myPLNMjMln3#2-scheduler)
  + [3. Kubelet](https://www.educative.io/courses/practical-guide-to-kubernetes/myPLNMjMln3#3-kubelet)
* [Sequential Breakdown of Events](https://www.educative.io/courses/practical-guide-to-kubernetes/myPLNMjMln3#sequential-breakdown-of-events)
* <https://medium.com/@kumargaurav1247/components-of-kubernetes-architecture-6feea4d5c712>

## Components Involved in a Pod’s Scheduling [#](https://www.educative.io/courses/practical-guide-to-kubernetes/myPLNMjMln3#components-involved-in-a-pods-scheduling)

Let’s discuss some of the details of Kubernetes components, and try to get an understanding of how Pod scheduling works.

**Three** major components were involved in the process:

### 1. API Server [#](https://www.educative.io/courses/practical-guide-to-kubernetes/myPLNMjMln3#1-api-server)

The API server is the central component of a Kubernetes cluster and it runs on the master node. Since we are using Minikube, both master and worker nodes are baked into the same virtual machine. However, a more serious Kubernetes cluster should have the two separated on different hosts.

All other components interact with API server and keep watch for changes. Most of the coordination in Kubernetes consists of a component writing to the API Server resource that another component is watching. The second component will then react to changes almost immediately.

### 2. Scheduler [#](https://www.educative.io/courses/practical-guide-to-kubernetes/myPLNMjMln3#2-scheduler)

The scheduler is also running on the master node. Its job is to watch for unassigned pods and assign them to a node which has available resources (CPU and memory) matching Pod requirements. Since we are running a single-node cluster, specifying resources would not provide much insight into their usage so we’ll leave them for later.

### 3. Kubelet [#](https://www.educative.io/courses/practical-guide-to-kubernetes/myPLNMjMln3#3-kubelet)

Kubelet runs on each node. Its primary function is to make sure that assigned pods are running on the node. It watches for any new Pod assignments for the node. If a Pod is assigned to the node Kubelet is running on, it will pull the Pod definition and use it to create containers through Docker or any other supported container engine.

## Sequential Breakdown of Events [#](https://www.educative.io/courses/practical-guide-to-kubernetes/myPLNMjMln3#sequential-breakdown-of-events)

The sequence of events that transpired with the kubectl create -f pod/db.yml command is as follows:

1. Kubernetes client (kubectl) sent a request to the API server requesting creation of a Pod defined in the pod/db.yml file.
2. Since the scheduler is watching the API server for new events, it detected that there is an unassigned Pod.
3. The scheduler decided which node to assign the Pod to and sent that information to the API server.
4. Kubelet is also watching the API server. It detected that the Pod was assigned to the node it is running on.
5. Kubelet sent a request to Docker requesting the creation of the containers that form the Pod. In our case, the Pod defines a single container based on the mongo image.
6. Finally, Kubelet sent a request to the API server notifying it that the Pod was created successfully.

The process might not make much sense right now since we are running a single-node cluster. If we had more VMs, scheduling might have happened somewhere else, and the complexity of the process would be easier to grasp. We’ll get there in due time.

The following illustration shows a Pod’s scheduling sequence.

Diagram

Description automatically generated

The above illustration shows us the sequence of events associated with a Pod’s scheduling.

# Single vs. Multi-Container Pods

In this lesson, we will briefly discuss a single-container and a multi-container Pod.

* [Why Single-container Pods?](#Why Single-container Pods? )
* [Multi-container Pods](#Multi-container Pods )

## Why Single-container Pods?

We should not think of Pods as resources that should do anything beyond a definition of the smallest unit in our cluster. A Pod is a collection of containers that share the same resources. Not much more. Everything else should be accomplished with higher-level constructs. We’ll explore how to scale Pods without changing their definition in one of the next chapters.

Let’s go back to our original multi-container Pod that defined api and db containers. That was a terrible design choice since it tightly couples one with the other. As a result, when we explore how to scale Pods (not containers), both would need to match. If, for example, we scale the Pod to three, we’d have three APIs and three DBs. Instead, we should have defined two Pods, one for each container (db and api). That would give us enough flexibility to treat each independently from the other.

There are quite a few other reasons not to put multiple containers in the same Pod. For now, just be patient. Most of the scenarios where you might think that multi-container Pod is a good solution will probably be solved through other resources.

📝 A **Pod** is a collection of containers. However, that does not mean that multi-container Pods are common. They are rare. Most Pods you’ll create will be single container units.

## Multi-container Pods

Does that mean that multi-container Pods are useless? They’re not. There are scenarios when having multiple containers in a Pod is a good idea. However, they are very specific and, in most cases, are based on one container that acts as the main service and the rest serving as side-cars.

A frequent use case is multi-container Pods used for:

* Continuous integration **(CI)**
* Continious Delivery **(CD)**
* Continuous Deployment processes **(CDP)**

We’ll explore them later. For now, we’ll focus on single-container Pods.

Let’s remove the Pod before we move onto container health.

1

kubectl delete -f pod/go-demo-2.yml

That’s it for now. In the next lesson, we will discuss monitoring the health of a Pod.

# Monitoring Health

In this lesson, we will find out why to monitor the health of services and how to achieve this using Kubernetes Probes.

**We'll cover the following**

* [Why to Monitor Health?](https://www.educative.io/courses/practical-guide-to-kubernetes/mEYX8n69GyG#why-to-monitor-health)
* [Kubernetes Probes](https://www.educative.io/courses/practical-guide-to-kubernetes/mEYX8n69GyG#kubernetes-probes)
  + [Liveness Probe](https://www.educative.io/courses/practical-guide-to-kubernetes/mEYX8n69GyG#liveness-probe)
  + [Readiness Probe](https://www.educative.io/courses/practical-guide-to-kubernetes/mEYX8n69GyG#readiness-probe)
* [Understanding the Updated Pod Definition](https://www.educative.io/courses/practical-guide-to-kubernetes/mEYX8n69GyG#understanding-the-updated-pod-definition)
* [Liveness Probe in Action](https://www.educative.io/courses/practical-guide-to-kubernetes/mEYX8n69GyG#liveness-probe-in-action)
* [Pods Are (Almost) Useless (By Themselves)](https://www.educative.io/courses/practical-guide-to-kubernetes/mEYX8n69GyG#pods-are-almost-useless-by-themselves)

## Why to Monitor Health? [#](https://www.educative.io/courses/practical-guide-to-kubernetes/mEYX8n69GyG#why-to-monitor-health)

The [go-demo-2](https://github.com/vfarcic/k8s-specs/blob/master/pod/go-demo-2.yml) Docker image is designed to fail on the first sign of trouble. In cases like that, there is no need for any health checks. When things go wrong:

* The main process stops.
* The container hosting the main process stops as well.
* Kubernetes restarts the failed container.

However, not all services are designed to fail fast. Even those that are might still benefit from additional health checks. For example, a back-end API can be up and running but, due to a memory leak, serves requests much slower than expected. Such a situation might benefit from a health check that would verify whether the service responds within, for example, two seconds.

## Kubernetes Probes [#](https://www.educative.io/courses/practical-guide-to-kubernetes/mEYX8n69GyG#kubernetes-probes)

We can exploit Kubernetes **liveness and readiness probes** for that.

### Liveness Probe [#](https://www.educative.io/courses/practical-guide-to-kubernetes/mEYX8n69GyG#liveness-probe)

livenessProbe can be used to confirm whether a container should be running. If the probe fails, Kubernetes will kill the container and apply restart policy which defaults to Always.

### Readiness Probe [#](https://www.educative.io/courses/practical-guide-to-kubernetes/mEYX8n69GyG#readiness-probe)

We’ll leave readinessProbe for [later](https://www.educative.io/collection/page/5376908829130752/4742963282313216/6332913870176256) since it is directly tied to Services.

Instead, we’ll explore livenessProbe. Both are defined in the same way so the experience with one of them can be easily applied to the other.

## Understanding the Updated Pod Definition [#](https://www.educative.io/courses/practical-guide-to-kubernetes/mEYX8n69GyG#understanding-the-updated-pod-definition)

Let’s take a look at an updated definition of the Pod we used thus far.

1

cat pod/go-demo-2-health.yml

The **output** is as follows.

Text

Description automatically generated

* **Line 8-12:** Don’t get confused by seeing two containers in this Pod. Those two should be defined in separate Pods. However, since that would require knowledge we are yet to obtain, and [go-demo-2](https://github.com/vfarcic/k8s-specs/blob/master/pod/go-demo-2.yml) doesn’t work without a database, we’ll have to stick with the example that specifies two containers. It won’t take long until we break it into pieces.
* **Line 16-19:** The additional definition is inside the livenessProbe. We defined that the action should be httpGet followed with the path and the port of the service. Since /this/path/does/not/exist is true to itself, the probe will fail, thus showing us what happens when a container is unhealthy. The host is not specified since it defaults to the Pod IP.
* **Line 20-23:** We declared that the first execution of the probe should be delayed by five seconds (initialDelaySeconds), that requests should timeout after two seconds (timeoutSeconds), that the process should be repeated every five seconds (periodSeconds), and (failureThreshold) define how many attempts it must try before giving up .

### Liveness Probe in Action [#](https://www.educative.io/courses/practical-guide-to-kubernetes/mEYX8n69GyG#liveness-probe-in-action)

Let’s take a look at the probe in action.

1

2

kubectl create \

    -f pod/go-demo-2-health.yml

We created the Pod with the probe. Now we must wait until the probe fails a few times. A minute is more than enough. Once we’re done waiting, we can describe the Pod.

1

2

kubectl describe \

    -f pod/go-demo-2-health.yml

The bottom of the **output** contains events. They are as follows.

...

Events:

Type Reason Age From Message

---- ------ ---- ---- -------

Normal Scheduled 6m default-scheduler Successfully assigned go-demo-2 to minikube

Normal SuccessfulMountVolume 6m kubelet, minikube MountVolume.SetUp succeeded for volume "default-token-7jc7q"

Normal Pulling 6m kubelet, minikube pulling image "mongo"

Normal Pulled 6m kubelet, minikube Successfully pulled image "mongo"

Normal Created 6m kubelet, minikube Created container

Normal Started 6m kubelet, minikube Started container

Normal Created 5m (x3 over 6m) kubelet, minikube Created container

Normal Started 5m (x3 over 6m) kubelet, minikube Started container

Warning Unhealthy 5m (x3 over 6m) kubelet, minikube Liveness probe failed: HTTP probe failed with statuscode: 404

Normal Pulling 5m (x4 over 6m) kubelet, minikube pulling image "vfarcic/go-demo-2"

Normal Killing 5m (x3 over 6m) kubelet, minikube Killing container with id docker://api:Container failed liveness probe.. Container will be killed and recreated.

Normal Pulled 5m (x4 over 6m) kubelet, minikube Successfully pulled image "vfarcic/go-demo-2"

We can see that, once the container started, the probe was executed, and that it failed. As a result, the container was killed only to be created again. In the output above, we can see that the process was repeated three times (3x over ...).

Please visit [Probe v1 core](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.18/#probe-v1-core) if you’d like to learn all the available options.

## Pods Are (Almost) Useless (By Themselves) [#](https://www.educative.io/courses/practical-guide-to-kubernetes/mEYX8n69GyG#pods-are-almost-useless-by-themselves)

Pods are fundamental building blocks in Kubernetes. In most cases, you will not create Pods directly. Instead, you’ll use higher level constructs like Controllers.

Pods are disposable. They are not long lasting services. Even though Kubernetes is doing its best to ensure that the containers in a Pod are (almost) always up-and-running, the same cannot be said for Pods. If a Pod fails, gets destroyed, or gets evicted from a Node, it will not be rescheduled. At least, not without a Controller. Similarly, if a whole node is destroyed, all the Pods on it will cease to exist. Pods do not heal by themselves. Excluding some special cases, Pods are not meant to be created directly.

📝 **Do not** create Pods by themselves. Let one of the controllers create Pods for you.

# 

# Getting Started with ReplicaSets

This lesson will introduce ReplicaSets to us and we will create a cluster for getting started with ReplicaSets.

* [Understanding](#_Understanding_ReplicaSets) ReplicaSets
* [Creating A Cluster](#_Creating_A_Cluster)

## Understanding ReplicaSets

Most applications should be scalable and all must be fault tolerant. Pods do not provide those features, **ReplicaSets** do.

We learned that Pods are the smallest unit in Kubernetes. We also learned that Pods are **not fault tolerant**. If a Pod is destroyed, Kubernetes will do nothing to remedy the problem. That is if Pods are created without **Controllers**.

The first Controller we’ll explore is called *ReplicaSet*. Its primary, and pretty much only function, is to ensure that a specified number of replicas of a Pod matches the actual state (almost) all the time. That means that ReplicaSets make Pods scalable.

We can think of ReplicaSets as a *self-healing* mechanism. As long as elementary conditions are met (e.g., enough memory and CPU), Pods associated with a ReplicaSet are guaranteed to run. They provide fault-tolerance and high availability.

If you’re familiar with Replication Controllers, it is worth mentioning that ReplicaSet is the next-generation *ReplicationController*. The only significant difference is that ReplicaSet has extended support for selectors. Everything else is the same. ReplicationController is considered deprecated, so we’ll focus only on ReplicaSet.

ReplicaSet’s primary function is to ensure that the specified number of replicas of a service are (almost) always running.

Let’s explore ReplicaSet through examples and see how it works and what it does.

The first step is to create a Kubernetes cluster.

All the commands from this chapter are available in the [04-rs.sh](https://gist.github.com/f6588da3d1c8a82100a81709295d4a93) Gist

## Creating A Cluster

We’ll continue using Minikube to simulate a cluster locally.

minikube start --vm-driver=virtualbox

kubectl config current-context

We created a single-node cluster and configured kubectl to use it.

Before we explore the first ReplicaSet example, we’ll enter into the local copy of the vfarcic[/k8s-spec](https://github.com/vfarcic/k8s-specs" \t "_blank) repository and pull the latest version.

cd k8s-specs

git pull

Now that the cluster is running and the repository with the specs is up-to-date, we can create our first ReplicaSet.

**Creating ReplicaSets**

In this lesson, first, we will create a ReplicaSet and then retrieve it.

* [Looking into the Definition](https://www.educative.io/courses/practical-guide-to-kubernetes/JQXlzjZRmMl#looking-into-the-definition)
* [Creating the ReplicaSet](https://www.educative.io/courses/practical-guide-to-kubernetes/JQXlzjZRmMl#creating-the-replicaset)

**Looking into the Definition**[#](https://www.educative.io/courses/practical-guide-to-kubernetes/JQXlzjZRmMl#looking-into-the-definition)

Let’s take a look at a ReplicaSet based on the Pod we created in the previous chapter.

cat rs/go-demo-2.yml

The **output** is as follows.

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: go-demo-2

spec:

replicas: 2

selector:

matchLabels:

type: backend

service: go-demo-2

template:

metadata:

labels:

type: backend

service: go-demo-2

db: mongo

language: go

spec:

containers:

- name: db

image: mongo:3.3

- name: api

image: vfarcic/go-demo-2

env:

- name: DB

value: localhost

livenessProbe:

httpGet:

path: /demo/hello

port: 8080

The apiVersion, kind, and metadata fields are mandatory with all Kubernetes objects. **ReplicaSet** is no exception, i.e., it is also a Kubernetes object.

* **Line 1:** We specified that the apiVersion is apps/v1.
* **Line 2-3:** The kind is ReplicaSet and metadata has the name key set to go-demo-2. We could have extended ReplicaSet metadata with labels. However, we skipped that part since they would serve only for informational purposes. They do not affect the behavior of the ReplicaSet.

You should be familiar with the above **three fields** since we already explored them when we worked with Pods. In addition to them, the spec section is mandatory as well.

* **Line 5-6:** The first field we defined in the spec section is replicas. It sets the desired number of replicas of the Pod. In this case, the ReplicaSet should ensure that two Pods should run concurrently. If we did not specify the value of the replicas, it would default to 1.
* **Line 7:** The next spec section is the selector. We use it to select which pods should be included in the ReplicaSet. It does not distinguish between the Pods created by a ReplicaSet or some other process. In other words, ReplicaSets and Pods are decoupled. If Pods that match the selector exist, ReplicaSet will do nothing. If they don’t, it will create as many Pods to match the value of the replicas field. Not only that ReplicaSet creates the Pods that are missing, but it also monitors the cluster and ensures that the desired number of replicas is (almost) always running. In case there are already more running Pods with the matching selector, some will be terminated to match the number set in replicas.
* **Line 8-10:** We used spec.selector.matchLabels to specify a few labels. They must match the labels defined in the spec.template. In our case, ReplicaSet will look for Pods with type set to backend and service set to go-demo-2. If Pods with those labels do not already exist, it’ll create them using the spec.template section.
* **Line 11-17:** The last section of the spec field is the template. It is the only required field in the spec, and it has the same schema as a Pod specification. At a minimum, the labels of the spec.template.metadata.labels section must match those specified in the spec.selector.matchLabels. We can set additional labels that will serve informational purposes only. ReplicaSet will make sure that the number of replicas matches the number of Pods with the same labels. In our case, we set type and service to the same values and added two additional ones (db and language). It might sound confusing that the spec.template.spec.containers field is mandatory. ReplicaSet will look for Pods with the matching labels created by other means. If we already created a Pod with labels type: backend and service: go-demo-2, this ReplicaSet would find them and would not create a Pod defined in spec.template. The main purpose of that field is to ensure that the desired number of replicas is running. If they are created by other means, ReplicaSet will do nothing. Otherwise, it’ll create them using the information in spec.template.
* **Line 18-23:** Finally, the spec.template.spec section contains the same containers definition we used in the previous chapter. It defines a Pod with two containers (db and api).

In the previous chapter, we claimed that those two containers should not belong to the same Pod. The same is true for the containers in Pods managed by the ReplicaSet. However, we did not yet have the opportunity to explore ways to allow containers running in different Pods to communicate with each other. So, for now, we’ll continue using the same *flawed Pods definition*.

**Creating the ReplicaSet**[#](https://www.educative.io/courses/practical-guide-to-kubernetes/JQXlzjZRmMl#creating-the-replicaset)

Let’s create the ReplicaSet and experience its advantages first hand.

1

kubectl create -f rs/go-demo-2.yml

We got the response that the replicaset "go-demo-2" was created. We can confirm that by listing all the ReplicaSets in the cluster.

1

kubectl get rs

The **output** is as follows.

NAME      DESIRED CURRENT READY AGE

go-demo-2 2       2       0     14s

We can see that the desired number of replicas is 2 and that it matches the current value. The value of the ready field is still 0 but, after the images are pulled, and the containers are running, it’ll change to 2.

Instead of retrieving all the replicas in the cluster, we can retrieve those specified in the rs/go-demo-2.yml file.

kubectl get -f rs/go-demo-2.yml

The **output** should be the same since, in both cases, there is only one ReplicaSet running inside the cluster.

All the other kubectl get arguments we explored in the previous chapter also apply to ReplicaSets or, to be more precise, to all Kubernetes objects. The same is true for kubectl describe command.

kubectl describe -f rs/go-demo-2.yml

The last lines of the **output** are as follows.

...

Events:

  Type   Reason           Age  From                  Message

  ----   ------           ---- ----                  -------

  Normal SuccessfulCreate 3m   replicaset-controller Created pod: go-demo-2-v59t5

  Normal SuccessfulCreate 3m   replicaset-controller Created pod: go-demo-2-5fd54

Judging by the events, we can see that ReplicaSet created two Pods while trying to match the desired state with the actual state.

Finally, if you are not yet convinced that the ReplicaSet created the missing Pods, we can list all those running in the cluster and confirm it.

kubectl get pods --show-labels

To be on the safe side, we used the --show-labels argument so that we can verify that the Pods in the cluster match those created by the ReplicaSet.

The **output** is as follows.

go-demo-2-5fd54 2/2   Running 0        6m  db=mongo,language=go,service=go-demo-2,type=backend

go-demo-2-v59t5 2/2   Running 0        6m  db=mongo,language=go,service=go-demo-2,type=backend

Diagram

Description automatically generated

The above illustration shows a ReplicaSet with two replicas of a Pod.

Till now, we have successfully replicated a Pod using ReplicaSets. In the next lesson, we will go through the sequential breakdown of this process.

# Sequential Breakdown of the Process

In this lesson, we will look into the occurrence of sequential events to create replicas of a Pod.

###### We'll cover the following

* [Sequential Breakdown of the Process](https://www.educative.io/courses/practical-guide-to-kubernetes/xVozZnlZNn3#sequential-breakdown-of-the-process)

## Sequential Breakdown of the Process [#](https://www.educative.io/courses/practical-guide-to-kubernetes/xVozZnlZNn3#sequential-breakdown-of-the-process)

The sequence of events that transpired with the kubectl create -f rs/go-demo-2.yml command is as follows.

1. Kubernetes client (kubectl) sent a request to the API server requesting the creation of a ReplicaSet defined in the rs/go-demo-2.yml file.
2. The controller is watching the API server for new events, and it detected that there is a new ReplicaSet object.
3. The controller creates two new pod definitions because we have configured replica value as 2 in rs/go-demo-2.yml file.
4. Since the scheduler is watching the API server for new events, it detected that there are two unassigned Pods.
5. The scheduler decided which node to assign the Pod and sent that information to the API server.
6. Kubelet is also watching the API server. It detected that the two Pods were assigned to the node it is running on.
7. Kubelet sent requests to Docker requesting the creation of the containers that form the Pod. In our case, the Pod defines two containers based on the mongo and api image. So in total four containers are created.
8. Finally, Kubelet sent a request to the API server notifying it that the Pods were created successfully.

Diagram

Description automatically generated

The sequence we described is useful when we want to understand everything that happened in the cluster from the moment we requested the creation of a new ReplicaSet. However, it might be too confusing so we’ll try to explain the same process through a diagram that more closely represents the cluster.

Diagram

Description automatically generated

📝 Typically, we’d have a multi-node cluster, and the Pods would be distributed across it. For now, while we’re using Minikube, there’s only one server that acts as both the master and the node. Later on, when we start working on multi-node clusters, the distribution of Pods will become evident. The same can be said for the architecture. We’ll explain different Kubernetes components in more detail later on.

In the next lesson, let’s see which types of operations we can perform on ReplicaSets.

# Operating ReplicaSets

In this lesson, we will explore the operating procedure of ReplicaSets and see its self-healing property in action.

* [Deleting ReplicaSets](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#deleting-replicasets)
* [Re-using the Same Pods](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#re-using-the-same-pods)
* [Updating the Definition](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#updating-the-definition)
* [Self-healing in Action](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#self-healing-in-action)
  + [Destroying a Pod](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#destroying-a-pod)
  + [Removing a label](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#removing-a-label)
  + [Re-adding the Label](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#re-adding-the-label)

## Deleting ReplicaSets [#](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#deleting-replicasets)

What would happen if we delete the ReplicaSet? As you might have guessed, both the ReplicaSet and everything it created (the Pods) would **disappear** with a single kubectl delete -f rs/go-demo-2.yml command.

However, since ReplicaSets and Pods are loosely coupled objects with matching labels, we can remove one **without deleting** the other.

We can, for example, remove the ReplicaSet we created while leaving the two Pods intact.

kubectl delete -f rs/go-demo-2.yml \

    --cascade=false

We used the --cascade=false argument to prevent Kubernetes from removing all the downstream objects. As a result, we got the confirmation that replicaset "go-demo-2" was deleted.

Let’s confirm that it is indeed removed from the system.

kubectl get rs

As expected, the **output** states that no resources were found.

If --cascade=false indeed prevents Kubernetes from removing the downstream objects, the Pods should continue running in the cluster. Let’s confirm the assumption.1

kubectl get pods

The **output** is as follows.

NAME            READY STATUS  RESTARTS AGE

go-demo-2-md5xp 2/2   Running 0        9m

go-demo-2-vnmf7 2/2   Running 0        9m

The **two Pods** created by the ReplicaSet are indeed still running in the cluster even though we removed the ReplicaSet.

The Pods that are currently running in the cluster do not have any relation with the ReplicaSet we created earlier. We deleted the ReplicaSet, and the Pods are still there.

Knowing that the ReplicaSet uses labels to decide whether the desired number of Pods is already running in the cluster, should lead us to the conclusion that if we create the same ReplicaSet again, it should reuse the two Pods that are running in the cluster. Let’s confirm that.

## Re-using the Same Pods [#](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#re-using-the-same-pods)

In addition to the kubectl create command we executed previously, we’ll also add the --save-config argument. It’ll save the configuration of the ReplicaSet thus allowing us to perform a few additional operations later on. We’ll get to them shortly. For now, the important thing is that we are about to create the same ReplicaSet we had before.

kubectl create -f rs/go-demo-2.yml \

    --save-config

The **output** states that the replicaset "go-demo-2" was created. Let’s see what happened with the Pods.

kubectl get pods

The **output** is as follows.

NAME            READY STATUS  RESTARTS AGE

go-demo-2-md5xp 2/2   Running 0        10m

go-demo-2-vnmf7 2/2   Running 0        10m

If you compare the names of the Pods, you’ll see that they are the **same as before** we created the ReplicaSet. It looked for matching labels, deduced that there are two Pods that match them, and decided that there’s no need to create new ones. The matching Pods fulfill the desired number of replicas.

## Updating the Definition [#](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#updating-the-definition)

Since we saved the configuration, we can apply an updated definition of the ReplicaSet. For example, we can use rs/go-demo-2-scaled.yml file that differs only in the number of replicas set to 4.

We could have created the ReplicaSet with apply in the first place, but we didn’t. The apply command automatically saves the configuration so that we can edit it later on. The create command does not do such thing by default so we had to save it with --save-config.

kubectl apply -f rs/go-demo-2-scaled.yml

This time, the **output** is slightly different. Instead of saying that the ReplicaSet was created, we can see that it was configured.

Let’s take a look at the Pods.

kubectl get pods

The **output** is as follows.

NAME            READY STATUS  RESTARTS AGE

go-demo-2-ckmtv 2/2   Running 0        50s

go-demo-2-lt4qm 2/2   Running 0        50s

go-demo-2-md5xp 2/2   Running 0        11m

go-demo-2-vnmf7 2/2   Running 0        11m

As expected, now there are **four Pods** in the cluster. If you pay closer attention to the names of the Pods, you’ll notice that two of them are the same as before.

When we applied the new configuration with replicas set to 4 instead of 2, Kubernetes updated the ReplicaSet which, in turn, evaluated the current state of the Pods with matching labels. It found two with the same labels and decided to create two more so that the new desired state can match the actual state.

## Self-healing in Action [#](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#self-healing-in-action)

We have already discussed that ReplicsSets have self-healing property. Let’s test this property by making a few changes to our system.

### Destroying a Pod [#](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#destroying-a-pod)

Let’s see what happens when a Pod is destroyed.

POD\_NAME=$(kubectl get pods -o name \

   | tail -1)

kubectl delete $POD\_NAME

We retrieved all the Pods and used -o name to retrieve only their names. The result was piped to tail -1 so that only one of the names is output. The result is stored in the environment variable POD\_NAME. The latter command used that variable to remove the Pod as a simulation of a failure.

Let’s take another look at the Pods in the cluster.

kubectl get pods

The **output** is as follows.

NAME              READY     STATUS        RESTARTS   AGE

go-demo-2-ckmtv   2/2       Running       0          10m

go-demo-2-lt4qm   2/2       Running       0          10m

go-demo-2-md5xp   2/2       Running       0          13m

go-demo-2-t8sfs   2/2       Running       0          30s

go-demo-2-vnmf7   0/2       Terminating   0          13m

We can see that the Pod we deleted is terminating. However, since we have a ReplicaSet with replicas set to 4, as soon as it discovered that the number of Pods dropped to 3, it created a new one. We just witnessed **self-healing** in action.

📝 We get the final output after the system goes through several stages so your output might differ from the above.

As long as there are enough available resources in the cluster, ReplicaSets will make sure that the specified number of Pod replicas are (almost) always up-and-running.

### Removing a label [#](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#removing-a-label)

Let’s see what happens if we remove one of the Pod labels ReplicaSet uses in its selector.

POD\_NAME=$(kubectl get pods -o name \

    | tail -1)

kubectl label $POD\_NAME service-

kubectl describe $POD\_NAME

We used the same command to retrieve the name of one of the Pods and executed the command that removed the label service.

ℹ️ Please note - at the end of the name of the label. It is the syntax that indicates that a label should be removed.

Finally, we described the Pod.

The **output** of the last command, limited to the labels section, is as follows.

...

Labels: db=mongo

        language=go

        type=backend

...

As you can see, the label service is gone.

Now, let’s list the Pods in the cluster and check whether there is any change.

kubectl get pods --show-labels

The **output** is as follows.

NAME            READY STATUS  RESTARTS AGE LABELS

go-demo-2-ckmtv 2/2   Running 0        24m db=mongo,language=go,service=go-demo-2,type=backend

go-demo-2-lt4qm 2/2   Running 0        24m db=mongo,language=go,service=go-demo-2,type=backend

go-demo-2-md5xp 2/2   Running 0        28m db=mongo,language=go,type=backend

go-demo-2-nrnbh 2/2   Running 0        4m  db=mongo,language=go,service=go-demo-2,type=backend

go-demo-2-t8sfs 2/2   Running 0        15m db=mongo,language=go,service=go-demo-2,type=backend

The total number of Pods increased to **five**. The moment we removed the service label from one of the Pods, the ReplicaSet discovered that the number of Pods matching the selector labels is three and created a new Pod.

Right now, we have four Pods controlled by the ReplicaSet and one running freely due to non-matching labels.

### Re-adding the Label [#](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#re-adding-the-label)

What would happen if we add the label we removed?

kubectl label $POD\_NAME service=go-demo-2

kubectl get pods --show-labels

We added the service=go-demo-2 label and listed all the Pods.

The **output** of the latter command is as follows.

NAME            READY STATUS      RESTARTS AGE LABELS

go-demo-2-ckmtv 2/2   Running     0        28m db=mongo,language=go,service=go-demo-2,type=backend

go-demo-2-lt4qm 2/2   Running     0        28m db=mongo,language=go,service=go-demo-2,type=backend

go-demo-2-md5xp 2/2   Running     0        31m db=mongo,language=go,service=go-demo-2,type=backend

go-demo-2-nrnbh 0/2   Terminating 0        7m  db=mongo,language=go,service=go-demo-2,type=backend

go-demo-2-t8sfs 2/2   Running     0        18m db=mongo,language=go,service=go-demo-2,type=backend

The moment we added the label, the ReplicaSet discovered that there are five Pods with matching selector labels. Since the specification states that there should be four replicas of the Pod, it removed one of the Pods so that the desired state matches the actual state.

The previous few examples showed, one more time, that ReplicaSets and Pods are **loosely coupled** through matching labels and that ReplicaSets are using those labels to maintain the parity between the actual and the desired state.

So far, self-healing worked as expected.

In the next lesson, we will go through a quick quiz to test our understanding of ReplicaSets.

# Operating ReplicaSets

In this lesson, we will explore the operating procedure of ReplicaSets and see its self-healing property in action.

###### We'll cover the following

* [Deleting ReplicaSets](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#deleting-replicasets)
* [Re-using the Same Pods](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#re-using-the-same-pods)
* [Updating the Definition](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#updating-the-definition)
* [Self-healing in Action](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#self-healing-in-action)
  + [Destroying a Pod](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#destroying-a-pod)
  + [Removing a label](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#removing-a-label)
  + [Re-adding the Label](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#re-adding-the-label)

## Deleting ReplicaSets [#](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#deleting-replicasets)

What would happen if we delete the ReplicaSet? As you might have guessed, both the ReplicaSet and everything it created (the Pods) would **disappear** with a single kubectl delete -f rs/go-demo-2.yml command.

However, since ReplicaSets and Pods are loosely coupled objects with matching labels, we can remove one **without deleting** the other.

We can, for example, remove the ReplicaSet we created while leaving the two Pods intact.

kubectl delete -f rs/go-demo-2.yml \

    --cascade=false

We used the --cascade=false argument to prevent Kubernetes from removing all the downstream objects. As a result, we got the confirmation that replicaset "go-demo-2" was deleted.

Let’s confirm that it is indeed removed from the system.

kubectl get rs

As expected, the **output** states that no resources were found.

If --cascade=false indeed prevents Kubernetes from removing the downstream objects, the Pods should continue running in the cluster. Let’s confirm the assumption.

kubectl get pods

The **output** is as follows.

NAME            READY STATUS  RESTARTS AGE

go-demo-2-md5xp 2/2   Running 0        9m

go-demo-2-vnmf7 2/2   Running 0        9m

The **two Pods** created by the ReplicaSet are indeed still running in the cluster even though we removed the ReplicaSet.

The Pods that are currently running in the cluster do not have any relation with the ReplicaSet we created earlier. We deleted the ReplicaSet, and the Pods are still there.

Knowing that the ReplicaSet uses labels to decide whether the desired number of Pods is already running in the cluster, should lead us to the conclusion that if we create the same ReplicaSet again, it should reuse the two Pods that are running in the cluster. Let’s confirm that.

## Re-using the Same Pods [#](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#re-using-the-same-pods)

In addition to the kubectl create command we executed previously, we’ll also add the --save-config argument. It’ll save the configuration of the ReplicaSet thus allowing us to perform a few additional operations later on. We’ll get to them shortly. For now, the important thing is that we are about to create the same ReplicaSet we had before.

kubectl create -f rs/go-demo-2.yml \

    --save-config

The **output** states that the replicaset "go-demo-2" was created. Let’s see what happened with the Pods.

kubectl get pods

The **output** is as follows.

NAME            READY STATUS  RESTARTS AGE

go-demo-2-md5xp 2/2   Running 0        10m

go-demo-2-vnmf7 2/2   Running 0        10m

If you compare the names of the Pods, you’ll see that they are the **same as before** we created the ReplicaSet. It looked for matching labels, deduced that there are two Pods that match them, and decided that there’s no need to create new ones. The matching Pods fulfill the desired number of replicas.

## Updating the Definition [#](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#updating-the-definition)

Since we saved the configuration, we can apply an updated definition of the ReplicaSet. For example, we can use rs/go-demo-2-scaled.yml file that differs only in the number of replicas set to 4.

We could have created the ReplicaSet with apply in the first place, but we didn’t. The apply command automatically saves the configuration so that we can edit it later on. The create command does not do such thing by default so we had to save it with --save-config.

kubectl apply -f rs/go-demo-2-scaled.yml

This time, the **output** is slightly different. Instead of saying that the ReplicaSet was created, we can see that it was configured.

Let’s take a look at the Pods.

kubectl get pods

The **output** is as follows.

NAME            READY STATUS  RESTARTS AGE

go-demo-2-ckmtv 2/2   Running 0        50s

go-demo-2-lt4qm 2/2   Running 0        50s

go-demo-2-md5xp 2/2   Running 0        11m

go-demo-2-vnmf7 2/2   Running 0        11m

As expected, now there are **four Pods** in the cluster. If you pay closer attention to the names of the Pods, you’ll notice that two of them are the same as before.

When we applied the new configuration with replicas set to 4 instead of 2, Kubernetes updated the ReplicaSet which, in turn, evaluated the current state of the Pods with matching labels. It found two with the same labels and decided to create two more so that the new desired state can match the actual state.

## Self-healing in Action [#](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#self-healing-in-action)

We have already discussed that ReplicsSets have self-healing property. Let’s test this property by making a few changes to our system.

### Destroying a Pod [#](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#destroying-a-pod)

Let’s see what happens when a Pod is destroyed.

POD\_NAME=$(kubectl get pods -o name \

   | tail -1)

kubectl delete $POD\_NAME

We retrieved all the Pods and used -o name to retrieve only their names. The result was piped to tail -1 so that only one of the names is output. The result is stored in the environment variable POD\_NAME. The latter command used that variable to remove the Pod as a simulation of a failure.

Let’s take another look at the Pods in the cluster.

kubectl get pods

The **output** is as follows.

NAME              READY     STATUS        RESTARTS   AGE

go-demo-2-ckmtv   2/2       Running       0          10m

go-demo-2-lt4qm   2/2       Running       0          10m

go-demo-2-md5xp   2/2       Running       0          13m

go-demo-2-t8sfs   2/2       Running       0          30s

go-demo-2-vnmf7   0/2       Terminating   0          13m

We can see that the Pod we deleted is terminating. However, since we have a ReplicaSet with replicas set to 4, as soon as it discovered that the number of Pods dropped to 3, it created a new one. We just witnessed **self-healing** in action.

📝 We get the final output after the system goes through several stages so your output might differ from the above.

As long as there are enough available resources in the cluster, ReplicaSets will make sure that the specified number of Pod replicas are (almost) always up-and-running.

### Removing a label [#](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#removing-a-label)

Let’s see what happens if we remove one of the Pod labels ReplicaSet uses in its selector.

POD\_NAME=$(kubectl get pods -o name \

    | tail -1)

kubectl label $POD\_NAME service-

kubectl describe $POD\_NAME

We used the same command to retrieve the name of one of the Pods and executed the command that removed the label service.

ℹ️ Please note - at the end of the name of the label. It is the syntax that indicates that a label should be removed.

Finally, we described the Pod.

The **output** of the last command, limited to the labels section, is as follows..

Labels: db=mongo

        language=go

        type=backend

...

As you can see, the label service is gone.

Now, let’s list the Pods in the cluster and check whether there is any change.

kubectl get pods --show-labels

The **output** is as follows.

NAME            READY STATUS  RESTARTS AGE LABELS

go-demo-2-ckmtv 2/2   Running 0        24m db=mongo,language=go,service=go-demo-2,type=backend

go-demo-2-lt4qm 2/2   Running 0        24m db=mongo,language=go,service=go-demo-2,type=backend

go-demo-2-md5xp 2/2   Running 0        28m db=mongo,language=go,type=backend

go-demo-2-nrnbh 2/2   Running 0        4m  db=mongo,language=go,service=go-demo-2,type=backend

go-demo-2-t8sfs 2/2   Running 0        15m db=mongo,language=go,service=go-demo-2,type=backend

The total number of Pods increased to **five**. The moment we removed the service label from one of the Pods, the ReplicaSet discovered that the number of Pods matching the selector labels is three and created a new Pod.

Right now, we have four Pods controlled by the ReplicaSet and one running freely due to non-matching labels.

### Re-adding the Label [#](https://www.educative.io/courses/practical-guide-to-kubernetes/qAL9KnxNMy3#re-adding-the-label)

What would happen if we add the label we removed?

kubectl label $POD\_NAME service=go-demo-2

kubectl get pods --show-labels

We added the service=go-demo-2 label and listed all the Pods.

The **output** of the latter command is as follows.

NAME            READY STATUS      RESTARTS AGE LABELS

go-demo-2-ckmtv 2/2   Running     0        28m db=mongo,language=go,service=go-demo-2,type=backend

go-demo-2-lt4qm 2/2   Running     0        28m db=mongo,language=go,service=go-demo-2,type=backend

go-demo-2-md5xp 2/2   Running     0        31m db=mongo,language=go,service=go-demo-2,type=backend

go-demo-2-nrnbh 0/2   Terminating 0        7m  db=mongo,language=go,service=go-demo-2,type=backend

go-demo-2-t8sfs 2/2   Running     0        18m db=mongo,language=go,service=go-demo-2,type=backend

The moment we added the label, the ReplicaSet discovered that there are five Pods with matching selector labels. Since the specification states that there should be four replicas of the Pod, it removed one of the Pods so that the desired state matches the actual state.

The previous few examples showed, one more time, that ReplicaSets and Pods are **loosely coupled** through matching labels and that ReplicaSets are using those labels to maintain the parity between the actual and the desired state.

So far, self-healing worked as expected.

In the next lesson, we will go through a quick quiz to test our understanding of ReplicaSets.

# Services:

# Getting Started with Communication

In this lesson, we will find out why we need to establish communication between Pods.

* [The Problem](https://www.educative.io/courses/practical-guide-to-kubernetes/BnwG01jKqoX#the-problem)
* [The Solution](#The Solution )
* [Creating A Cluster](#Creating A Cluster )

## The Problem [#](https://www.educative.io/courses/practical-guide-to-kubernetes/BnwG01jKqoX#the-problem)

Pods are the smallest unit in Kubernetes and have a relatively short life-span. They are born, and they are destroyed. They are never healed. The system heals itself by creating new Pods (cells) and by terminating those that are unhealthy or those that are surplus. The system is long-living, Pods are not.

Controllers, together with other components like the scheduler, are making sure that the Pods are doing the right thing. They control the scheduler. We used only one of them so far.

ReplicaSet is in charge of making sure that the desired number of Pods is always running. If there’s too few of them, new ones will be created. If there’s too many of them, some will be destroyed. Pods that become unhealthy are terminated as well. All that, and a bit more, is controlled by ReplicaSet.

The problem with our current setup is that there are no communication paths. Our Pods cannot speak with each other. So far, only containers inside a Pod can talk with each other through localhost. That led us to the design where both the API and the database needed to be inside the same Pod. That was a lousy solution for quite a few reasons.

The main problem is that we cannot scale one without the other. We could not design the setup in a way that there are, for example, three replicas of the API and one replica of the database. The primary obstacle was communication.

Truth be told, each Pod does get its own address. We could have split the API and the database into different Pods and configure the API Pods to communicate with the database through the address of the Pod it lives in.

However, since Pods are unreliable, short-lived, and volatile, we cannot assume that the database would always be accessible through the IP of a Pod. When that Pod gets destroyed (or fails), the ReplicaSet would create a new one and assign it a new address.

We need a stable, never-to-be-changed address that will forward requests to whichever Pod is currently running.

## The Solution [#](https://www.educative.io/courses/practical-guide-to-kubernetes/BnwG01jKqoX#the-solution)

Kubernetes Services provide addresses through which associated Pods can be accessed.

Let’s see Services in action.

ℹ️ All the commands from this chapter are available in the [05-svc.sh](https://gist.github.com/ae2527a1e960ec3fea19adb00aab6fd7) Gist.

## [Creating](#The Problem ) A Cluster [#](https://www.educative.io/courses/practical-guide-to-kubernetes/BnwG01jKqoX#creating-a-cluster)

You know the drill. Every chapter starts by pulling the latest code from the [vfarcic/k8s-specs](https://github.com/vfarcic/k8s-specs" \t "_blank) repository, and with the creation of a new Minikube cluster.

cd k8s-specs

git pull

minikube start --vm-driver=virtualbox

kubectl config current-context

Now we have the latest code pulled and the Minikube cluster is running (again).

# Creating Services by Exposing Ports

In this lesson, we will explore how to create Kubernetes Services by exposing ports.

* [Creating ReplicaSets](#Creating ReplicaSets )
* [Exposing a Resource](#Exposing a Resource )
* [Other Types of Services](#Other Types of Services ) 
  + ClusterIP
  + LoadBalancer
  + ExternalName

## Creating ReplicaSets [#](https://www.educative.io/courses/practical-guide-to-kubernetes/gkWZzJmpkOj#creating-replicasets)

Before we dive into services, we should create a ReplicaSet similar to the one we used in the previous chapter. It’ll provide the Pods we can use to demonstrate how Services work.

Let’s take a quick look at the ReplicaSet definition.

cat svc/go-demo-2-rs.yml

The only significant difference is the db container definition. It is as follows.

...

- name: db

  image: mongo:3.3

  command: ["mongod"]

  args: ["--rest", "--httpinterface"]

  ports:

  - containerPort: 28017

    protocol: TCP

...

We customized the command and the arguments so that MongoDB exposes the REST interface. We also defined the containerPort. Those additions are needed so that we can test that the database is accessible through the Service.

Let’s create the ReplicaSet.

kubectl create -f svc/go-demo-2-rs.yml

kubectl get -f svc/go-demo-2-rs.yml

We created the ReplicaSet and retrieved its state from Kubernetes. The **output** is as follows.

NAME      DESIRED CURRENT READY AGE

go-demo-2 2       2       2     1m

You might need to wait until both replicas are up-and-running. If, in your case, the READY column does not yet have the value 2, please wait for a while and get the state again. We can proceed after both replicas are running.

## Exposing a Resource [#](https://www.educative.io/courses/practical-guide-to-kubernetes/gkWZzJmpkOj#exposing-a-resource)

We can use the kubectl expose command to expose a resource as a new Kubernetes Service. That resource can be a Deployment, another Service, a ReplicaSet, a ReplicationController, or a Pod. We’ll expose the ReplicaSet since it is already running in the cluster.

kubectl expose rs go-demo-2 \

    --name=go-demo-2-svc \

    --target-port=28017 \

    --type=NodePort

* **Line 1:** We specified that we want to expose a ReplicaSet (rs).
* **Line 2:** The name of the new Service should be go-demo-2-svc.
* **Line 3:** The port that should be exposed is 28017 (the port MongoDB interface is listening to).
* **Line 4:** we specified that the type of the Service should be NodePort.

As a result, the target port will be exposed on every node of the cluster to the outside world, and it will be routed to one of the Pods controlled by the ReplicaSet.

## Other Types of Services [#](https://www.educative.io/courses/practical-guide-to-kubernetes/gkWZzJmpkOj#other-types-of-services)

There are other Service types we could have used to establish communication:

### ClusterIP [#](https://www.educative.io/courses/practical-guide-to-kubernetes/gkWZzJmpkOj#clusterip)

ClusterIP (the default type) exposes the port only inside the cluster. Such a port would not be accessible from anywhere outside. ClusterIP is useful when we want to enable communication between Pods and still prevent any external access.

If NodePort is used, ClusterIP will be created automatically.

### LoadBalancer [#](https://www.educative.io/courses/practical-guide-to-kubernetes/gkWZzJmpkOj#loadbalancer)

The LoadBalancer type is only useful when combined with cloud provider’s load balancer.

### ExternalName [#](https://www.educative.io/courses/practical-guide-to-kubernetes/gkWZzJmpkOj#externalname)

ExternalName maps a service to an external address (e.g., kubernetes.io).

In this chapter, we’ll focus on NodePortand ClusterIP types. LoadBalancer will have to wait until we move our cluster to one of the cloud providers and ExternalName has a very limited usage.

# Sequential Breakdown of the Process

In this lesson, we will go through the sequential processes kicked off by a Service creation.

* [The Sequence](#The Sequence )
  + 📝 A note to the Windows users

## The Sequence [#](https://www.educative.io/courses/practical-guide-to-kubernetes/3YQ85oJnAKn#the-sequence)

The processes that were initiated with the creation of the Service are as follows:

1. Kubernetes client (kubectl) sent a request to the API server requesting the creation of the Service based on Pods created through the go-demo-2 ReplicaSet.
2. Endpoint controller is watching the API server for new service events. It detected that there is a new Service object.
3. Endpoint controller created endpoint objects with the same name as the Service, and it used Service selector to identify endpoints (in this case the IP and the port of go-demo-2 Pods).
4. kube-proxy is watching for service and endpoint objects. It detected that there is a new Service and a new endpoint object.
5. kube-proxy added iptables rules which capture traffic to the Service port and redirect it to endpoints. For each endpoint object, it adds iptables rule which selects a Pod.
6. The kube-dns add-on is watching for Service. It detected that there is a new service.
7. The kube-dns added db's record to the dns server (skydns).

The sequence of events followed by request to create a Service

The sequence we described is useful when we want to understand everything that happened in the cluster from the moment we requested the creation of a new Service. However, it might be too confusing so we’ll try to explain the same process through a diagram that more closely represents the cluster.

The Kubernetes components view when requesting creation of a Service

Let’s take a look at our new Service.

kubectl describe svc go-demo-2-svc

The **output** is as follows.

Name: go-demo-2-svc

Namespace: default

Labels: db=mongo

language=go

service=go-demo-2

type=backend

Annotations: <none>

Selector: service=go-demo-2,type=backend

Type: NodePort

IP: 10.0.0.194

Port: <unset> 28017/TCP

TargetPort: 28017/TCP

NodePort: <unset> 31879/TCP

Endpoints: 172.17.0.4:28017,172.17.0.5:28017

Session Affinity: None

External Traffic Policy: Cluster

Events: <none>

* **Line 1-2:** We can see the name and the namespace. We did not yet explore namespaces (coming up later) and, since we didn’t specify any, it is set to default.
* **Line 3-6:** Since the Service is associated with the Pods created through the ReplicaSet, it inherited all their labels. The selector matches the one from the ReplicaSet. The Service is not directly associated with the ReplicaSet (or any other controller) but with Pods through matching labels.
* **Line 9-13:** Next is the NodePort type which exposes ports to all the nodes. Since NodePort automatically created ClusterIP type as well, all the Pods in the cluster can access the TargetPort. The Port is set to 28017. That is the port that the Pods can use to access the Service. Since we did not specify it explicitly when we executed the command, its value is the same as the value of the TargetPort, which is the port of the associated Pod that will receive all the requests. NodePort was generated automatically since we did not set it explicitly. It is the port which we can use to access the Service and, therefore, the Pods from outside the cluster. In most cases, it should be randomly generated, that way we avoid any clashes.

Let’s see whether the Service indeed works.

PORT=$(kubectl get svc go-demo-2-svc \

-o jsonpath="{.spec.ports[0].nodePort}")

IP=$(minikube ip)

open "http://$IP:$PORT"

### 📝 A note to the Windows users [#](https://www.educative.io/courses/practical-guide-to-kubernetes/3YQ85oJnAKn#a-note-to-the-windows-users)

Git Bash might not be able to use the open command. If that’s the case, replace the open command with echo. As a result, you’ll get the full address that should be opened directly in your browser of choice.

* **Line 1-2:** We used the filtered output of the kubectl get command to retrieve the nodePort and store it as the environment variable PORT.
* **Line 3:** We retrieved the IP of the minikube VM.
* **Line 4:** Finally, We opened MongoDB UI in a browser through the service port.

The Service created by exposing the ReplicaSet

As it has been already mentioned in the previous chapters, creating Kubernetes objects using imperative commands is **not a good idea** unless we’re trying some quick hack.

The same applies to Services. Even though kubectl expose did the work, we should try to use a documented approach through YAML files. In that spirit, we’ll destroy the service we created and start over.

kubectl delete svc go-demo-2-svc

Now that we have destroyed the Service, we will explore creating Services through declarative syntax in the next lesson.

# Creating Services through Declarative Syntax

In this lesson, we will learn to create Services through declarative syntax.

* [Looking into the Syntax](#Looking into the Syntax )
* [Creating the Service](#Creating the Service )
* [Request Forwarding](#Request Forwarding )
* [Now We Can Split](#Now We Can Split )
* [Destroying Everything](#Destroying Everything )

## Looking into the Syntax [#](https://www.educative.io/courses/practical-guide-to-kubernetes/JPWOmVJw002#looking-into-the-syntax)

We can accomplish a similar result as the one using kubectl expose through the svc/go-demo-2-svc.yml specification.

cat svc/go-demo-2-svc.yml

The **output** is as follows.

apiVersion: v1

kind: Service

metadata:

  name: go-demo-2

spec:

  type: NodePort

  ports:

  - port: 28017

    nodePort: 30001

    protocol: TCP

  selector:

    type: backend

    service: go-demo-2

* **Line 1-4:** You should be familiar with the meaning of apiVersion, kind, and metadata, so we’ll jump straight into the spec section.
* **Line 5:** Since we already explored some of the options through the kubectl expose command, the spec should be relatively easy to grasp.
* **Line 6:** The type of the Service is set to NodePort meaning that the ports will be available both within the cluster as well as from outside by sending requests to any of the nodes.
* **Line 7-10:** The ports section specifies that the requests should be forwarded to the Pods on port 28017. The nodePort is new. Instead of letting the service expose a random port, we set it to the explicit value of 30001. Even though, in most cases, that is not a good practice, I thought it might be a good idea to demonstrate that option as well. The protocol is set to TCP. The only other alternative would be to use UDP. We could have skipped the protocol altogether since TCP is the default value but, sometimes, it is a good idea to leave things as a reminder of an option.
* **Line 11-13:** The selector is used by the Service to know which Pods should receive requests. It works in the same way as ReplicaSet selectors. In this case, we defined that the service should forward requests to Pods with labels type set to backend and service set to go-demo. Those two labels are set in the Pods spec of the ReplicaSet.

## Creating the Service [#](https://www.educative.io/courses/practical-guide-to-kubernetes/JPWOmVJw002#creating-the-service)

Now that there’s no mystery in the definition, we can proceed and create the Service.

kubectl create -f svc/go-demo-2-svc.yml

kubectl get -f svc/go-demo-2-svc.yml

We created the Service and retrieved its information from the API server. The **output** of the latter command is as follows.

NAME      TYPE     CLUSTER-IP EXTERNAL-IP PORT(S)         AGE

go-demo-2 NodePort 10.0.0.129 <none>      28017:30001/TCP 10m

Now that the Service is running (again), we can double-check that it is working as expected by trying to access MongoDB UI.

open "http://$IP:30001"

Since we fixed the nodePort to 30001, we did not have to retrieve the Port from the API server. Instead, we used the IP of the Minikube node and the hard-coded port 30001 to open the UI.

Diagram

Description automatically generated

Let’s take a look at the endpoint. It holds the list of Pods that should receive requests.

kubectl get ep go-demo-2 -o yaml

The **output** is as follows.

apiVersion: v1

kind: Endpoints

metadata:

  creationTimestamp: 2017-12-12T16:00:51Z

  name: go-demo-2

  namespace: default

  resourceVersion: "5196"

  selfLink: /api/v1/namespaces/default/endpoints/go-demo-2

  uid: a028b9a7-df55-11e7-a8ef-080027d94e34

subsets:

- addresses:

  - ip: 172.17.0.4

    nodeName: minikube

    targetRef:

      kind: Pod

      name: go-demo-2-j8kdw

      namespace: default

      resourceVersion: "5194"

      uid: ac70f868-df4d-11e7-a8ef-080027d94e34

  - ip: 172.17.0.5

    nodeName: minikube

    targetRef:

      kind: Pod

      name: go-demo-2-5vlcc

      namespace: default

      resourceVersion: "5184"

      uid: ac7214d9-df4d-11e7-a8ef-080027d94e34

  ports:

We can see that there are two subsets, corresponding to the two Pods that contain the same labels as the Service selector.

## Request Forwarding [#](https://www.educative.io/courses/practical-guide-to-kubernetes/JPWOmVJw002#request-forwarding)

Each Pod has a unique IP that is included in the algorithm used when forwarding requests. Actually, it’s not much of an algorithm. Requests will be sent to those Pods randomly. That randomness results in something similar to round-robin load balancing. If the number of Pods does not change, each will receive an approximately equal number of requests.

Random requests forwarding should be enough for most use cases. If it’s not, we’d need to resort to a third-party solution. However soon, when the newer Kubernetes versions get released, we’ll have an alternative to the iptables solution. We’ll be able to apply different types of load balancing algorithms like last connection, destination hashing, newer queue, and so on. Still, the current solution is based on iptables, and we’ll stick with it, for now.

## Now We Can Split [#](https://www.educative.io/courses/practical-guide-to-kubernetes/JPWOmVJw002#now-we-can-split)

So far, we have repeated a few times that our current Pod design is flawed. We have two containers (an API and a database) packaged together. This prevents us from scaling one without the other. Now that we learned how to use Services, we can redesign our Pod solution.

## Destroying Everything [#](https://www.educative.io/courses/practical-guide-to-kubernetes/JPWOmVJw002#destroying-everything)

Before we move on, we’ll delete the Service and the ReplicaSet we created.

kubectl delete -f svc/go-demo-2-svc.yml

kubectl delete -f svc/go-demo-2-rs.yml

Both the ReplicaSet and the Service are gone, and we can start anew.

In the next lesson, we will split the Pods and establish communication between them through Services.

# Splitting the Pod and Establishing Communication through Services

In this lesson, we will split up the Pods, create a separate DB pod and a Service to communicate with it.

* [Looking into the Definition](#Looking into the Definition )
* [Creating the ReplicaSet](#Creating the ReplicaSet )
* [Creating the Service](#Creating the Service )

## Looking into the Definition [#](https://www.educative.io/courses/practical-guide-to-kubernetes/7AJqjny2nNO#looking-into-the-definition)

Let’s take a look at a ReplicaSet definition for a Pod with only the database.

cat svc/go-demo-2-db-rs.yml

The **output** is as follows.

apiVersion: apps/v1

kind: ReplicaSet

metadata:

  name: go-demo-2-db

spec:

  selector:

    matchLabels:

      type: db

      service: go-demo-2

  template:

    metadata:

      labels:

        type: db

        service: go-demo-2

        vendor: MongoLabs

    spec:

      containers:

      - name: db

        image: mongo:3.3

        ports:

        - containerPort: 28017

We’ll comment only on the things that changed.

Since this ReplicaSet defines only the database, we reduced the number of replicas to 1. Truth be told, MongoDB should be scaled as well, but that’s out of the scope of this chapter. For now, we’ll pretend that one replica of a database is enough.

Since selector labels need to be unique, we changed them slightly. The service is still go-demo-2, but the type was changed to db.

The rest of the definition is the same except that the containers now contain only mongo. We’ll define the API in a separate ReplicaSet.

## Creating the ReplicaSet [#](https://www.educative.io/courses/practical-guide-to-kubernetes/7AJqjny2nNO#creating-the-replicaset)

Let’s create the ReplicaSet before we move to the Service that will reference its Pod.

kubectl create \

-f svc/go-demo-2-db-rs.yml

One object was created, three are left to go.

## Creating the Service [#](https://www.educative.io/courses/practical-guide-to-kubernetes/7AJqjny2nNO#creating-the-service)

The next one is the Service for the Pod we just created through the ReplicaSet.

cat svc/go-demo-2-db-svc.yml

The **output** is as follows.

apiVersion: v1

kind: Service

metadata:

name: go-demo-2-db

spec:

ports:

- port: 27017

selector:

type: db

service: go-demo-2

This Service definition does not contain anything new.

* There is no type, so it’ll default to ClusterIP.
* Since there is no reason for anyone outside the cluster to communicate with the database, there’s no need to expose it using the NodePort type.
* We also skipped specifying the NodePort, since only internal communication within the cluster is allowed.
* The same is true for the protocol. TCP is all we need, and it happens to be the default one.
* Finally, the selector labels are the same as the labels that define the Pod.

Let’s create the Service.

kubectl create \

-f svc/go-demo-2-db-svc.yml

We are finished with the database. The ReplicaSet will make sure that the Pod is (almost) always up-and-running and the Service will allow other Pods to communicate with it through a fixed DNS.

# Creating the Split API Pods

In this lesson, we will create API Pods using ReplicaSet and establish communication by creating Service.

* Looking into the Definition
* The readinessProbe
* Creating the ReplicaSet
* Creating the Service
* Accessing the API
* Destroying Services

## Looking into the Definition [#](https://www.educative.io/courses/practical-guide-to-kubernetes/xooK68rq4nq#looking-into-the-definition)

Moving to the backend API…

cat svc/go-demo-2-api-rs.yml

The **output** is as follows.

spec:

  replicas: 3

  selector:

    matchLabels:

      type: api

      service: go-demo-2

  template:

    metadata:

      labels:

        type: api

        service: go-demo-2

        language: go

    spec:

      containers:

      - name: api

        image: vfarcic/go-demo-2

        env:

        - name: DB

          value: go-demo-2-db

        readinessProbe:

          httpGet:

            path: /demo/hello

            port: 8080

          periodSeconds: 1

        livenessProbe:

          httpGet:

            path: /demo/hello

            port: 8080

Just as with the database, this ReplicaSet should be familiar since it’s very similar to the one we used before. We’ll comment only on the differences.

* **Line 6:** The number of replicas is set to 3. That solves one of the main problems we had with the previous ReplicaSets that defined Pods with both containers. Now the number of replicas can differ, and we have one Pod for the database, and three for the backend API.
* **Line 14:** In the labels section, type label is set to api so that both the ReplicaSet and the (soon to come) Service can distinguish the Pods from those created for the database.
* **Line 22-23:** We have the environment variable DB set to go-demo-2-db. The code behind the vfarcic/go-demo-2 image is written in a way that the connection to the database is established by reading that variable. In this case, we can say that it will try to connect to the database running on the DNS go-demo-2-db. If you go back to the database Service definition, you’ll notice that its name is go-demo-2-db as well. If everything works correctly, we should expect that the DNS was created with the Service and that it’ll forward requests to the database.

## The readinessProbe [#](https://www.educative.io/courses/practical-guide-to-kubernetes/xooK68rq4nq#the-readinessprobe)

The readinessProbe should be used as an indication that the service is ready to serve requests. When combined with Services construct, only containers with the readinessProbe state set to Success will receive requests.

In earlier Kubernetes versions it used userspace proxy mode. Its advantage is that the proxy would retry failed requests to another Pod. With the shift to the iptables mode, that feature is lost. However, iptables are much faster and more reliable, so the loss of the retry mechanism is well compensated. That does not mean that the requests are sent to Pods “blindly”. The lack of the retry mechanism is mitigated with readinessProbe, which we added to the ReplicaSet.

The readinessProbe has the same fields as the livenessProbe. We used the same values for both, except for the periodSeconds, where instead of relying on the default value of 10, we set it to 1.

While livenessProbe is used to determine whether a Pod is alive or it should be replaced by a new one, the readinessProbe is used by the iptables. A Pod that does not pass the readinessProbe will be excluded and will not receive requests. In theory, requests might be still sent to a faulty Pod, between two iterations. Still, such requests will be small in number since the iptables will change as soon as the next probe responds with HTTP code less than 200, or equal or greater than 400.

## Creating the ReplicaSet [#](https://www.educative.io/courses/practical-guide-to-kubernetes/xooK68rq4nq#creating-the-replicaset)

Let’s create the ReplicaSet.

kubectl create \

    -f svc/go-demo-2-api-rs.yml

## Creating the Service [#](https://www.educative.io/courses/practical-guide-to-kubernetes/xooK68rq4nq#creating-the-service)

Only one object is missing, that is Service.

cat svc/go-demo-2-api-svc.yml

The **output** is as follows.

apiVersion: v1

kind: Service

metadata:

  name: go-demo-2-api

spec:

  type: NodePort

  ports:

  - port: 8080

  selector:

    type: api

    service: go-demo-2

There’s nothing truly new in this definition. The type is set to NodePort since the API should be accessible from outside the cluster. The selector label type is set to api so that it matches the labels defined for the Pods.

That is the last object we’ll create (in this section), so let’s move on and do it.

kubectl create \

    -f svc/go-demo-2-api-svc.yml

We’ll take a look at what we have in the cluster.

kubectl get all

The **output** is as follows.

NAME                      READY   STATUS    RESTARTS   AGE

pod/go-demo-2-api-r55fs   1/1     Running   0          3m32s

pod/go-demo-2-api-sng48   1/1     Running   0          3m32s

pod/go-demo-2-api-vvcbp   1/1     Running   0          3m32s

pod/go-demo-2-db-bwvkb    1/1     Running   0          4m20s

NAME                    TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE

service/go-demo-2-api   NodePort    10.110.71.67    <none>        8080:31148/TCP   3m23s

service/go-demo-2-db    ClusterIP   10.104.40.176   <none>        27017/TCP        4m1s

service/kubernetes      ClusterIP   10.96.0.1       <none>        443/TCP          16m

NAME                            DESIRED   CURRENT   READY   AGE

replicaset.apps/go-demo-2-api   3         3         3       3m33s

replicaset.apps/go-demo-2-db    1         1         1       4m20s

Both ReplicaSets for db and api are there, followed by the three replicas of the go-demo-2-api Pods and one replica of the go-demo-2-db Pod. Finally, the two Services are running as well, together with the one created by Kubernetes itself.

## Accessing the API [#](https://www.educative.io/courses/practical-guide-to-kubernetes/xooK68rq4nq#accessing-the-api)

Before we proceed, it might be worth mentioning that the code behind the vfarcic/go-demo-2 image is designed to fail if it cannot connect to the database. The fact that the three replicas of the go-demo-2-api Pod are running means that the communication is established. The only verification left is to check whether we can access the API from outside the cluster.

Let’s try that out.

PORT=$(kubectl get svc go-demo-2-api \

    -o jsonpath="{.spec.ports[0].nodePort}")

curl -i "http://$IP:$PORT/demo/hello"

We retrieved the port of the service (we still have the Minikube node IP from before) and used it to send a request.

The **output** of the last command is as follows.

HTTP/1.1 200 OK

Date: Tue, 12 Dec 2017 21:27:51 GMT

Content-Length: 14

Content-Type: text/plain; charset=utf-8

hello, world!

We got the response 200 and a friendly hello, world! message indicating that the API is indeed accessible from outside the cluster.

## Destroying Services [#](https://www.educative.io/courses/practical-guide-to-kubernetes/xooK68rq4nq#destroying-services)

Before we move further, we’ll delete the objects we created.

kubectl delete -f svc/go-demo-2-db-rs.yml

kubectl delete -f svc/go-demo-2-db-svc.yml

kubectl delete -f svc/go-demo-2-api-rs.yml

kubectl delete -f svc/go-demo-2-api-svc.yml

Everything we created is gone, and we can start over.

At this point, you might be wondering whether it is overkill to have four YAML files for a single application. Can’t we simplify the definitions? Not really. Can we define everything in a single file? Read the next lesson.

# Defining Multiple Objects in the Same YAML file

In this lesson, we will define all the objects used thus far in a single YAML file.

###### We'll cover the following

* + [Defining Multiple Objects In The Same YAML file](https://www.educative.io/courses/practical-guide-to-kubernetes/m2Wv13KEJ7A#defining-multiple-objects-in-the-same-yaml-file)

## Defining Multiple Objects In The Same YAML file [#](https://www.educative.io/courses/practical-guide-to-kubernetes/m2Wv13KEJ7A#defining-multiple-objects-in-the-same-yaml-file)

The vfarcic/go-demo-2 and mongo images form the same stack. They work together and having four YAML definitions is confusing. It would get even more confusing later on since we are going to add more objects to the stack. Things would be much simpler and easier if we would move all the objects we created thus far into a single YAML definition. Fortunately, that is very easy to accomplish.

Let’s take a look at yet another YAML file.

1

cat svc/go-demo-2.yml

We won’t display the output since it is the same as the contents of the previous four YAML files combined. The only difference is that each object definition is separated by three dashes (---).

Let’s create the objects defined in that file.

kubectl create -f svc/go-demo-2.yml

kubectl get -f svc/go-demo-2.yml

The **output** of the latter command is as follows.

NAME DESIRED CURRENT READY AGE

replicaset.apps/go-demo-2-db 1 1 1 11s

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

service/go-demo-2-db ClusterIP 10.103.63.57 <none> 27017/TCP 11s

NAME DESIRED CURRENT READY AGE

replicaset.apps/go-demo-2-api 3 3 2 11s

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

service/go-demo-2-api NodePort 10.101.92.202 <none> 8080:30643/TCP 11s

The two ReplicaSets and the two Services were created, and we can rejoice in replacing four files with one.

Finally, to be on the safe side, we’ll also double check that the stack API is up-and-running and accessible.

PORT=$(kubectl get svc go-demo-2-api \

-o jsonpath="{.spec.ports[0].nodePort}")

curl -i "http://$IP:$PORT/demo/hello"

The response is 200 indicating that everything works as expected.

# Discovering Services

In this lesson, we will go through the discovery process of Services.

* [Discovering Services](https://www.educative.io/courses/practical-guide-to-kubernetes/39vvkrY048x#discovering-services)
* [Sequential Breakdown of the Process](https://www.educative.io/courses/practical-guide-to-kubernetes/39vvkrY048x#sequential-breakdown-of-the-process)

## Discovering Services [#](https://www.educative.io/courses/practical-guide-to-kubernetes/39vvkrY048x#discovering-services)

Services can be discovered through two principal modes:

* Environment variables
* DNS

Every Pod gets environment variables for each of the active Services. They are provided in the same format as what Docker links expect, as well with the simpler Kubernetes-specific syntax.

Let’s take a look at the environment variables available in one of the Pods we’re running.

POD\_NAME=$(kubectl get pod \

    --no-headers \

    -o=custom-columns=NAME:.metadata.name \

    -l type=api,service=go-demo-2 \

    | tail -1)

kubectl exec $POD\_NAME env

The **output**, limited to the environment variables related to the go-demo-2-db service, is as follows.

GO\_DEMO\_2\_DB\_PORT=tcp://10.0.0.250:27017

GO\_DEMO\_2\_DB\_PORT\_27017\_TCP\_ADDR=10.0.0.250

GO\_DEMO\_2\_DB\_PORT\_27017\_TCP\_PROTO=tcp

GO\_DEMO\_2\_DB\_PORT\_27017\_TCP\_PORT=27017

GO\_DEMO\_2\_DB\_PORT\_27017\_TCP=tcp://10.0.0.250:27017

GO\_DEMO\_2\_DB\_SERVICE\_HOST=10.0.0.250

GO\_DEMO\_2\_DB\_SERVICE\_PORT=27017

The first five variables are using the Docker format. If you already worked with Docker networking, you should be familiar with them. At least, if you’re familiar with the way Swarm (standalone) and Docker Compose operate. Later version of Swarm (Mode) still generate the environment variables but they are mostly abandoned by the users in favour of DNSes.

The last two environment variables are Kubernetes specific and follow the [SERVICE\_NAME]\_SERVICE\_HOST and [SERVICE\_NAME]\_SERIVCE\_PORT format (service name is upper-cased).

No matter which set of environment variables you choose to use (if any), they all serve the same purpose. They provide a reference we can use to connect to a Service and, therefore to the related Pods.

Things will become more evident when we describe the go-demo-2-db Service.

kubectl describe svc go-demo-2-db

The **output** is as follows.

Name:              go-demo-2-db

Namespace:         default

Labels:            <none>

Annotations:       <none>

Selector:          service=go-demo-2,type=db

Type:              ClusterIP

IP:                10.0.0.250

Port:              <unset>  27017/TCP

TargetPort:        27017/TCP

Endpoints:         172.17.0.4:27017

Session Affinity:  None

Events:            <none>

The key is in the IP field. That is the IP through which this service can be accessed and it matches the values of the environment variables GO\_DEMO\_2\_DB\_\* and GO\_DEMO\_2\_DB\_SERVICE\_HOST.

The code inside the containers that form the go-demo-2-api Pods could use any of those environment variables to construct a connection string towards the go-demo-2-db Pods. For example, we could have used GO\_DEMO\_2\_DB\_SERVICE\_HOST to connect to the database. And, yet, we didn’t do that. The reason is simple. It is easier to use DNS instead.

Let’s take another look at the snippet from the go-demo-2-api-rs.yml ReplicaSet definition.

cat svc/go-demo-2-api-rs.yml

The **output** limited to the environment variable is as follows.

...

env:

- name: DB

  value: go-demo-2-db

...

We declared an environment variable with the name of the Service (go-demo-2-db). That variable is used by the code as a connection string to the database.

Kubernetes converts Service names into DNSes and adds them to the DNS server. It is a cluster add-on that is already set up by Minikube.

## Sequential Breakdown of the Process [#](https://www.educative.io/courses/practical-guide-to-kubernetes/39vvkrY048x#sequential-breakdown-of-the-process)

Let’s go through the sequence of events related to service discovery and components involved.

1. When the api container go-demo-2 tries to connect with the go-demo-2-db Service, it looks at the nameserver configured in /etc/resolv.conf. kubelet configured the nameserver with the kube-dns Service IP (10.96.0.10) during the Pod scheduling process.
2. The container queries the DNS server listening to port 53. go-demo-2-db DNS gets resolved to the service IP 10.0.0.19. This DNS record was added by kube-dns during the service creation process.
3. The container uses the service IP which forwards requests through the iptables rules. They were added by kube-proxy during Service and Endpoint creation process.
4. Since we only have one replica of the go-demo-2-db Pod, iptables forwards requests to just one endpoint. If we had multiple replicas, iptables would act as a load balancer and forward requests randomly among Endpoints of the Service.

Diagram

Description automatically generated

# 

# Deployments: Getting Started with Deploying Releases

In this lesson, we will discuss releasing features to production with zero downtime.

* Deploying Releases
* Why Zero Downtime?
* Kubernetes Deployments
* Creating A Cluster

## Deploying Releases [#](https://www.educative.io/courses/practical-guide-to-kubernetes/7DKG6rKzLor#deploying-releases)

In today’s competitive environment, we have to release features to production as soon as they are developed and tested. The need for frequent releases fortifies the need for zero-downtime deployments.

We learned how to deploy our applications packaged as Pods, how to scale them through ReplicaSets, and how to enable communication through Services. However, all that is useless if we cannot update those applications with new releases. That is where Kubernetes Deployments come in handy.

The desired state of our applications is changing all the time. The most common reasons for new states are new releases. The process is relatively simple. We make a change and commit it to a code repository. We build it, and we test it. Once we’re confident that it works as expected, we deploy it to a cluster.

It does not matter whether that deployment is to a development, test, staging, or production environment. We need to deploy a new release to a cluster, even when that is a single-node Kubernetes running on a laptop. No matter how many environments we have, the process should always be the same or, at least, as similar as possible.

## Why Zero Downtime? [#](https://www.educative.io/courses/practical-guide-to-kubernetes/7DKG6rKzLor#why-zero-downtime)

The deployment must produce no downtime. It does not matter whether it is performed on a testing or a production cluster. Interrupting consumers is disruptive, and that leads to loss of money and confidence in a product.

Gone are the days when users did not care if an application sometimes did not work. There are so many competitors out there that a single bad experience might lead users to another solution. With today’s scale, 0.1% of failed requests is considered disastrous.

While we might never be able to reach 100% availability, we should certainly not cause downtime ourselves and must minimize other factors that could cause downtime.

## Kubernetes Deployments [#](https://www.educative.io/courses/practical-guide-to-kubernetes/7DKG6rKzLor#kubernetes-deployments)

Kubernetes Deployments provide us with the tools we need to avoid such failures by allowing us to update our applications without downtime.

Let’s explore how Kubernetes Deployments work and the benefits we gain by adopting them.

ℹ️ All the commands from this chapter are available in the [06-deploy.sh](https://gist.github.com/677a0d688f65ceb01e31e33db59a4400) Gist.

## Creating A Cluster [#](https://www.educative.io/courses/practical-guide-to-kubernetes/7DKG6rKzLor#creating-a-cluster)

Creating a cluster at the beginning of each chapter allows us to jump into any part of the course without worrying whether there is a requirement to meet from previous chapters. It also allows us to pause between chapters without stressing our laptops by running a VM that is not in use.

Let’s get it over with.

cd k8s-specs

git pull

minikube start --vm-driver=virtualbox

kubectl config current-context

The code was updated, the cluster is up-and-running, and we can start exploring Deployments.

# Deploying New Releases

In this lesson, we will go through the Kubernetes deployment definition and will create a deployment.

* [Looking into the Definition](https://www.educative.io/courses/practical-guide-to-kubernetes/g2qYlwyvGpl#looking-into-the-definition)
* [Creating the Deployment](https://www.educative.io/courses/practical-guide-to-kubernetes/g2qYlwyvGpl#creating-the-deployment)
* [Describing the Deployment](https://www.educative.io/courses/practical-guide-to-kubernetes/g2qYlwyvGpl#describing-the-deployment)

Just as we are not supposed to create Pods directly but using other controllers like ReplicaSet, we are not supposed to create ReplicaSets either. Kubernetes Deployments will create them for us. If you’re wondering why is this so? You’ll have to wait a little while longer to find out.

First, we’ll create a few Deployments and, once we are familiar with the process and the outcomes, it’ll become obvious why they are better at managing ReplicaSets than we are.

## Looking into the Definition [#](https://www.educative.io/courses/practical-guide-to-kubernetes/g2qYlwyvGpl#looking-into-the-definition)

Let’s take a look at a Deployment specification for the database ReplicaSet we’ve been using thus far.

cat deploy/go-demo-2-db.yml

The **output** is as follows.

apiVersion: apps/v1

kind: Deployment

metadata:

  name: go-demo-2-db

spec:

  selector:

    matchLabels:

      type: db

      service: go-demo-2

  template:

    metadata:

      labels:

        type: db

        service: go-demo-2

        vendor: MongoLabs

    spec:

      containers:

      - name: db

        image: mongo:3.3

        ports:

        - containerPort: 28017

If you compare this Deployment with the ReplicaSet we created in the previous chapter, you’ll probably have a hard time finding a difference. Apart from the kind field, they are the same.

Since, in this case, both the Deployment and the ReplicaSet are the same, you might be wondering what the advantage of using one over the other is.

ℹ️ We will regularly add --record to the kubectl create commands. This allows us to track each change to our resources such as a Deployments.

## Creating the Deployment [#](https://www.educative.io/courses/practical-guide-to-kubernetes/g2qYlwyvGpl#creating-the-deployment)

Let’s create the Deployment and explore what it offers.

kubectl create \

-f deploy/go-demo-2-db.yml \

--record

kubectl get -f deploy/go-demo-2-db.yml

The **output** of the latter command is as follows.

NAME READY UP-TO-DATE AVAILABLE AGE

go-demo-2-db 0/1 1 0 4s

## Describing the Deployment [#](https://www.educative.io/courses/practical-guide-to-kubernetes/g2qYlwyvGpl#describing-the-deployment)

The Deployment was created. However, get does not provide us much info, so let’s describe it.

kubectl describe \

-f deploy/go-demo-2-db.yml

The **output**, limited to the last few lines, is as follows.

...

Events:

Type Reason Age From Message

---- ------ ---- ---- -------

Normal ScalingReplicaSet 2m deployment-controller Scaled up replica set go-demo-2-db-75fbcbb5cd to 1

From the Events section, we can observe that the Deployment created a ReplicaSet. Or, to be more precise, that it scaled it. That is interesting.

It shows that Deployments control ReplicaSets. The Deployment created the ReplicaSet which, in turn, created Pods.

Let’s confirm that by retrieving the list of all the objects.

kubectl get all

The **output** is as follows.

NAME READY STATUS RESTARTS AGE

pod/go-demo-2-db-694bfb44cb-n6rxl 1/1 Running 0 7m49s

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 9m40s

NAME READY UP-TO-DATE AVAILABLE AGE

deployment.apps/go-demo-2-db 1/1 1 1 7m49s

NAME DESIRED CURRENT READY AGE

replicaset.apps/go-demo-2-db-694bfb44cb 1 1 1 7m49s

All three objects were created, and you might be wondering why we created the Deployment at all. You might think that we’d have the same result if we created a ReplicaSet directly. You’d be right.

So far, from the functional point of view, there is no difference between a ReplicaSet created directly or using a Deployment.

The real advantage of Deployments becomes evident if we try to change some of its aspects. For example, we might choose to upgrade MongoDB to version 3.4.

The following figure summarizes the cascading effect of deployments resulting in the creation of pods, containers, and replicaSets.

Diagram

Description automatically generated

Deployment and its cascading effect that creates a ReplicaSet and, through it, Pods

In the next lesson, we will go through the sequential breakdown of the process of creating deployment.

# Sequential Breakdown of the Process

In this lesson, we will go through the sequential breakdown of the Kubernetes Deployment process.

* [The Sequence](https://www.educative.io/courses/practical-guide-to-kubernetes/gxBA6PkoGlD#the-sequence)

## The Sequence [#](https://www.educative.io/courses/practical-guide-to-kubernetes/gxBA6PkoGlD#the-sequence)

Before we move onto Deployment updates, we’ll go through our usual ritual of seeing the process through a sequence diagram. We won’t repeat the explanation of the events that happened after the ReplicaSet object was created as those steps were already explained in the previous chapters.

1. Kubernetes client (kubectl) sent a request to the API server requesting the creation of a Deployment defined in the deploy/go-demo-2-db.yml file.
2. The deployment controller is watching the API server for new events, and it detected that there is a new Deployment object.
3. The deployment controller creates a new ReplicaSet object.
4. Diagram

   Description automatically generated with medium confidence

The sequence of events followed by request to create a deployment

The above illustration is self-explanatory with the sequence of the processes linked to the deployment process.

# Updating Deployments

In this lesson, we will learn to update the Kubernetes Deployments.

* [Updating the db Image](https://www.educative.io/courses/practical-guide-to-kubernetes/gk1jYnRor4Z#updating-the-db-image)
* [Describing the Deployment](https://www.educative.io/courses/practical-guide-to-kubernetes/gk1jYnRor4Z#describing-the-deployment)
* [Looking into the Cluster](https://www.educative.io/courses/practical-guide-to-kubernetes/gk1jYnRor4Z#looking-into-the-cluster)
* [Exploring Ways to Update Deployment](https://www.educative.io/courses/practical-guide-to-kubernetes/gk1jYnRor4Z#exploring-ways-to-update-deployment)
  + [Updating Using Commands](https://www.educative.io/courses/practical-guide-to-kubernetes/gk1jYnRor4Z#updating-using-commands)
  + [Updating the YAML File](https://www.educative.io/courses/practical-guide-to-kubernetes/gk1jYnRor4Z#updating-the-yaml-file)
* [Finishing off by Adding a Service](https://www.educative.io/courses/practical-guide-to-kubernetes/gk1jYnRor4Z#finishing-off-by-adding-a-service)

## Updating the db Image [#](https://www.educative.io/courses/practical-guide-to-kubernetes/gk1jYnRor4Z#updating-the-db-image)

Let’s see what happens when we set a new image to the db Pod.

kubectl set image \

    -f deploy/go-demo-2-db.yml \

    db=mongo:3.4 \

    --record

It’ll take a while until the new image is pulled.

## Describing the Deployment [#](https://www.educative.io/courses/practical-guide-to-kubernetes/gk1jYnRor4Z#describing-the-deployment)

Once it’s done, we can describe the Deployment by checking the events it created.

kubectl describe \

    -f deploy/go-demo-2-db.yml

The last few lines of the **output** are as follows.

...

Events:

  Type    Reason             Age   From                   Message

  ----    ------             ----  ----                   -------

  Normal  ScalingReplicaSet  19m   deployment-controller  Scaled up replica set go-demo-2-db-75fbcbb5cd to 1

  Normal  ScalingReplicaSet  5m    deployment-controller  Scaled up replica set go-demo-2-db-f8d4b86ff to 1

  Normal  ScalingReplicaSet  0s    deployment-controller  Scaled down replica set go-demo-2-db-75fbcbb5cd to 0

We can see that it created a new ReplicaSet and that it scaled the old ReplicaSet to 0. If, in your case, the last line did not appear, you’ll need to wait until the new version of the mongo image is pulled.

Instead of operating directly on the level of Pods, the Deployment created a new ReplicaSet which, in turn, produced Pods based on the new image. Once they became fully operational, it scaled the old ReplicaSet to 0.

Since we are running a ReplicaSet with only one replica, it might not be clear why it used that strategy. When we create a Deployment for the API, things will become more evident.

## Looking into the Cluster [#](https://www.educative.io/courses/practical-guide-to-kubernetes/gk1jYnRor4Z#looking-into-the-cluster)

To be on the safe side, we might want to retrieve all the objects from the cluster.

kubectl get all

The **output** is as follows.

NAME                                READY   STATUS              RESTARTS   AGE

pod/go-demo-2-db-694bfb44cb-n6rxl   1/1     Running             0          17m

pod/go-demo-2-db-6b97cd9dfc-kbzp4   0/1     ContainerCreating   0          47s

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE

service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   19m

NAME                           READY   UP-TO-DATE   AVAILABLE   AGE

deployment.apps/go-demo-2-db   1/1     1            1           17m

NAME                                      DESIRED   CURRENT   READY   AGE

replicaset.apps/go-demo-2-db-694bfb44cb   1         1         1       17m

replicaset.apps/go-demo-2-db-6b97cd9dfc   1         1         0       48s

As you can see, both ReplicaSets are there. However, one is inactive (scaled to 0).

You’ll notice that contained within the name of the Pod is a hash which matches the hash in the name of the new ReplicaSet, namely f8d4b86ff. Even though it might look like it is a random value, it is not.

If you destroy the Deployment and create it again, you’ll notice that the hash in the Pod name and ReplicaSet name remain consistent. This value is generated by hashing the PodTemplate of the ReplicaSet. As long as the PodTemplate is the same, the hash value will be the same as well. That way a Deployment can know whether anything related to the Pods has changed and, if it does, will create a new ReplicaSet.

## Exploring Ways to Update Deployment [#](https://www.educative.io/courses/practical-guide-to-kubernetes/gk1jYnRor4Z#exploring-ways-to-update-deployment)

### Updating Using Commands [#](https://www.educative.io/courses/practical-guide-to-kubernetes/gk1jYnRor4Z#updating-using-commands)

The kubectl set image command is not the only way to update a Deployment. We could also have used kubectl edit as well.

The command would be as follows.

kubectl edit -f deploy/go-demo-2-db.yml

⚠️ **Please do NOT execute it.** If you do, you’ll need to type :q followed by the enter key to exit.

The above edit command is not a good way to update the definition. It is unpractical and undocumented. The kubectl set image is more useful if we’d like to integrate Deployment updates with one of the CI/CD tools.

Since we’ll have a chapter dedicated to continuous deployment, we’ll continue using kubectl set image for now.

### Updating the YAML File [#](https://www.educative.io/courses/practical-guide-to-kubernetes/gk1jYnRor4Z#updating-the-yaml-file)

Another alternative would be to update the YAML file and execute the kubectl apply command. While that is a good idea for applications that do not update frequently, it does not fit well with those that change weekly, daily, or even hourly.

MongoDB is one of those that might get updated with a new release only a couple of times a year so having an always up-to-date YAML file in your source code repository is an excellent practice.

## Finishing off by Adding a Service [#](https://www.educative.io/courses/practical-guide-to-kubernetes/gk1jYnRor4Z#finishing-off-by-adding-a-service)

We used kubectl set image just as a way to introduce you to what’s coming next when we explore frequent deployments without downtime.

A simple update of Pod images is far from what Deployment offers. To see its real power, we should deploy the API. Since it can be scaled to multiple Pods, it’ll provide us with a much better playground.

Before we move on, let’s finish with the database by adding a Service and, therefore, enabling internal cluster communication to it.

kubectl create \

    -f deploy/go-demo-2-db-svc.yml \

    --record

Now that we have added the service, in the next lesson, we will learn how to deploy with zero down-time.

# Defining a Zero-Downtime Deployment

In this lesson, we will look into the definition of a zero-downtime deployment.

* [Looking into the Definition](https://www.educative.io/courses/practical-guide-to-kubernetes/qVqox9zrg8D#looking-into-the-definition)
* [Deployment Strategies](https://www.educative.io/courses/practical-guide-to-kubernetes/qVqox9zrg8D#deployment-strategies)
  + [Recreate Strategy](https://www.educative.io/courses/practical-guide-to-kubernetes/qVqox9zrg8D#recreate-strategy)
  + [RollingUpdate Strategy](https://www.educative.io/courses/practical-guide-to-kubernetes/qVqox9zrg8D#rollingupdate-strategy)
* [The Template](https://www.educative.io/courses/practical-guide-to-kubernetes/qVqox9zrg8D#the-template)

Updating a single-replica MongoDB cannot demonstrate true power behind Deployments. We need a scalable service. It’s not that MongoDB cannot be scaled (it can), but it is not as straight-forward as an application that was designed to be scalable. We’ll jump to the second application in the stack and create a Deployment of the ReplicaSet that will create Pods based on the vfarcic/go-demo-2 image.

Zero-downtime deployment is a prerequisite for higher frequency releases.

## Looking into the Definition [#](https://www.educative.io/courses/practical-guide-to-kubernetes/qVqox9zrg8D#looking-into-the-definition)

Let’s take a look at the Deployment definition of the API.

cat deploy/go-demo-2-api.yml

The **output** is as follows.

apiVersion: apps/v1

kind: Deployment

metadata:

  name: go-demo-2-api

spec:

  replicas: 3

  selector:

    matchLabels:

      type: api

      service: go-demo-2

  minReadySeconds: 1

  progressDeadlineSeconds: 60

  revisionHistoryLimit: 5

  strategy:

    type: RollingUpdate

    rollingUpdate:

      maxSurge: 1

      maxUnavailable: 1

  template:

    metadata:

      labels:

        type: api

        service: go-demo-2

        language: go

    spec:

      containers:

      - name: api

        image: vfarcic/go-demo-2

We’ll skip explaining apiVersion, kind, and metadata, since they always follow the same pattern.

* **Line 5-7:** The spec section has a few of the fields we haven’t seen before, and a few of those we are familiar with. The replicas and the selector are the same as what we used in the ReplicaSet from the previous chapter.
* **Line 11:** minReadySeconds defines the minimum number of seconds before Kubernetes starts considering the Pods healthy. We put the value of this field to 1 second. The default value is 0, meaning that the Pods will be considered available as soon as they are ready and, when specified, livenessProbe returns OK. If in doubt, omit this field and leave it to the default value of 0. We defined it mostly for demonstration purposes.
* **Line 13:** The next field is revisionHistoryLimit. It defines the number of old ReplicaSets we can rollback. Like most of the fields, it is set to the sensible default value of 10. We changed it to 5 and, as a result, we will be able to rollback to any of the previous five ReplicaSets.
* **Line 14:** The strategy can be either the RollingUpdate or the Recreate type. The latter will kill all the existing Pods before an update. Recreate resembles the processes we used in the past when the typical strategy for deploying a new release was first to stop the existing one and then put a new one in its place. This approach inevitably leads to downtime. The only case when this strategy is useful is when applications are not designed for two releases to coexist. Unfortunately, that is still more common than it should be. If you’re in doubt whether your application is like that, ask yourself the following question. Would there be an adverse effect if two different versions of my application are running in parallel? If that’s the case, a Recreate strategy might be a good choice and you must be aware that you cannot accomplish zero-downtime deployments.

## Deployment Strategies [#](https://www.educative.io/courses/practical-guide-to-kubernetes/qVqox9zrg8D#deployment-strategies)

Let’s look into a bit of detail of both Recreate and RollingUpdate strategies.

### Recreate Strategy [#](https://www.educative.io/courses/practical-guide-to-kubernetes/qVqox9zrg8D#recreate-strategy)

The Recreate strategy is much better suited for our single-replica database. We should have set up the native database replication (not the same as Kubernetes ReplicaSet object), but, that is out of the scope of this chapter.

If we’re running the database as a single replica, we must have mounted a network drive volume. That would allow us to avoid data loss when updating it or in case of a failure. Since most databases (MongoDB included) cannot have multiple instances writing to the same data files, killing the old release before creating a new one is a good strategy when replication is absent. We’ll apply it later.

### RollingUpdate Strategy [#](https://www.educative.io/courses/practical-guide-to-kubernetes/qVqox9zrg8D#rollingupdate-strategy)

The RollingUpdate strategy is the default type, for a good reason. It allows us to deploy new releases without downtime. It creates a new ReplicaSet with zero replicas and, depending on other parameters, increases the replicas of the new one, and decreases those from the old one. The process is finished when the replicas of the new ReplicaSet entirely replace those from the old one.

When RollingUpdate is the strategy of choice, it can be fine-tuned with the maxSurge and maxUnavailable fields. The former defines the maximum number of Pods that can exceed the desired number (set using replicas). It can be set to an absolute number (e.g., 2) or a percentage (e.g., 35%). The total number of Pods will never exceed the desired number (set using replicas) and the maxSurge combined. The default value is 25%.

maxUnavailable defines the maximum number of Pods that are not operational. If, for example, the number of replicas is set to 15 and this field is set to 4, the minimum number of Pods that would run at any given moment would be 11. Just as the maxSurge field, this one also defaults to **25%**. If this field is not specified, there will always be at least **75%** of the desired Pods.

In most cases, the default values of the Deployment specific fields are a good option. We changed the default settings only as a way to demonstrate better all the options we can use. We’ll remove them from most of the Deployment definitions that follow.

## The Template [#](https://www.educative.io/courses/practical-guide-to-kubernetes/qVqox9zrg8D#the-template)

The template is the same PodTemplate we used before. A best practice is to be explicit with image tags like we did when we set mongo:3.3. However, that might not always be the best strategy with the images we’re building. Given we employ right practices, we can rely on latest tags being stable. Even if we discover they’re not, we can remedy that quickly by creating a new latest tag. However, We cannot expect the same from third-party images. They must always be tagged to a specific version.

⚠️ Never deploy third-party images based on latest tags. By being explicit with the release, we have more control over what is running in production, as well as what should be the next upgrade.

We won’t always use latest for our services, but only for the initial Deployments. Assuming that we are doing our best to maintain the latest tag stable and production-ready, it is handy when setting up the cluster for the first time. After that, each new release will be with a specific tag. Our automated continuous deployment pipeline will do that for us in one of the next chapters.

# Creating a Zero-Downtime Deployment

In this lesson, we will create a zero-downtime deployment using the file defined in the previous lesson.

* [Creating Deployment](https://www.educative.io/courses/practical-guide-to-kubernetes/m7VkMqXPzzA#creating-deployment)

## Creating Deployment [#](https://www.educative.io/courses/practical-guide-to-kubernetes/m7VkMqXPzzA#creating-deployment)

Before we explore rolling updates, we should create the Deployment and, with it, the first release of our application.

kubectl create \

    -f deploy/go-demo-2-api.yml \

    --record

kubectl get -f deploy/go-demo-2-api.yml

We created the Deployment and retrieved the object from the Kubernetes API server.

The **output** of the latter command is as follows.

NAME          DESIRED  UP-TO-DATE AVAILABLE AGE

go-demo-2-api 3             3          3         1m

Please make sure that the number of available Pods is 3. Wait for a few moments, if that’s not the case. Once all the Pods are up-and-running, we’ll have a Deployment that created a new ReplicaSet which, in turn, created three Pods based on the latest release of the vfarcic/go-demo-2 image.

Let’s see what happens when we set a new image.

kubectl set image \

    -f deploy/go-demo-2-api.yml \

    api=vfarcic/go-demo-2:2.0 \

    --record

There are a few ways we can observe what is happening during the update. One of those is through the kubectl rollout status command.

kubectl rollout status -w \

    -f deploy/go-demo-2-api.yml

The **output** is as follows.

...

deployment "go-demo-2-api" successfully rolled out

From the last entry, we can see that the rollout of the new deployment was successful. Depending on the time that passed between setting the new image and displaying the rollout status, you might have seen other entries marking the progress. However, I think that the events from the kubectl describe command are painting a better picture of the process that was executed.

kubectl describe \

    -f deploy/go-demo-2-api.yml

The last lines of the **output** are as follows.

...

Replicas: 3 desired | 3 updated | 3 total | 3 available | 0 unavailable

...

OldReplicaSets:  <none>

NewReplicaSet:   go-demo-2-api-68c75f4f5 (3/3 replicas created)

Events:

  Type   Reason            Age  From                  Message

  ----   ------            ---- ----                  -------

  Normal ScalingReplicaSet 2m   deployment-controller Scaled up replica set go-demo-2-api-68df567fb5 to 3

  Normal ScalingReplicaSet 2m   deployment-controller Scaled up replica set go-demo-2-api-68c75f4f5 to 1

  Normal ScalingReplicaSet 2m   deployment-controller Scaled down replica set go-demo-2-api-68df567fb5 to 2

  Normal ScalingReplicaSet 2m   deployment-controller Scaled up replica set go-demo-2-api-68c75f4f5 to 2

  Normal ScalingReplicaSet 2m   deployment-controller Scaled down replica set go-demo-2-api-68df567fb5 to 1

  Normal ScalingReplicaSet 2m   deployment-controller Scaled up replica set go-demo-2-api-68c75f4f5 to 3

  Normal ScalingReplicaSet 2m   deployment-controller Scaled down replica set go-demo-2-api-68df567fb5 to 0

We can see that the number of desired replicas is 3. The same number was updated and all are available.

At the bottom of the output are events associated with the Deployment. The process started by increasing the number of replicas of the new ReplicaSet (go-demo-2-api-68c75f4f5) to 1. Next, it decreased the number of replicas of the old ReplicaSet (go-demo-2-api-68df567fb5) to 2. The same process of increasing replicas of the new, and decreasing replicas of the old ReplicaSet continued until the new one got the desired number (3), and the old one dropped to zero.

There was no downtime throughout the process. Users would receive a response from the application no matter whether they sent it before, during, or after the update. The only important thing is that, during the update, a response might have come from the old or the new release. During the update process, both releases were running in parallel.

Let’s take a look at the rollout history.

kubectl rollout history \

    -f deploy/go-demo-2-api.yml

The **output** is as follows.

deployments "go-demo-2-api"

REVISION CHANGE-CAUSE

1        kubectl create --filename=deploy/go-demo-2-api.yml --record=true

2        kubectl set image api=vfarcic/go-demo-2:2.0 --filename=deploy/go-demo-2-api.yml

We can see that, so far, there were two revisions of the software. The change cause shows which command created each of those revisions.

How about ReplicaSets?

kubectl get rs

The **output**, limited to go-demo-2-api, is as follows.

NAME                     DESIRED CURRENT READY AGE

go-demo-2-api-68c75f4f5  3       3       3     4m

go-demo-2-api-68df567fb5 0       0       0     4m

...

We can see that the Deployment did not modify the ReplicaSet, but that it created a new one and, at the end of the process, the old one was scaled to zero replicas.

Graphical user interface, application

Description automatically generated

Deployment controller rolling update workflow

The above diagram illustrates the workflow of the Deployment controller rolling update.

# Rolling Back or Rolling Forward?

In this lesson, we will discuss different scenarios to help us decide whether to roll back the Deployment or roll forward.

* + [Understanding the Scenarios](https://www.educative.io/courses/practical-guide-to-kubernetes/x1wL3zynOYE#understanding-the-scenarios)
  + [Rolling Back with Kubernetes](https://www.educative.io/courses/practical-guide-to-kubernetes/x1wL3zynOYE#rolling-back-with-kubernetes)
  + [Getting the Rollout History](https://www.educative.io/courses/practical-guide-to-kubernetes/x1wL3zynOYE#getting-the-rollout-history)

## Understanding the Scenarios [#](https://www.educative.io/courses/practical-guide-to-kubernetes/x1wL3zynOYE#understanding-the-scenarios)

At this point, we are, more or less, capable of deploying new releases to production as soon as they are ready. However, there will be problems. Something unexpected will happen. A bug will sneak in and put our production cluster at risk. What should we do in such a case? The answer to that question largely depends on the size of the changes and the frequency of deployments.

If we are using continuous deployment process, we are deploying new releases to production fairly often. Instead of waiting until features accumulate, we are deploying small chunks. In such cases, fixing a problem might be just as fast as rolling back.

After all, how much time would it take you to fix a problem caused by only a few hours of work (maybe a day) and that was discovered minutes after you committed? Probably not much. The problem was introduced by a very recent change that is still in the engineer’s head. Fixing it should not take long, and we should be able to deploy a new release soon.

You might not have frequent releases, or the amount of changes included is more than a couple of hundreds of lines of code. In such a case, rolling forward might not be as fast as it should be. Still, rolling back might not even be possible.

We might not be able to revert the deployment if database schema changed, and it is not compatible with the previous versions of the back-end that uses it. The moment the first transaction enters, we might lose the option to roll-back. At least, not without losing the data generated since the new release.

ℹ️ Rolling back a release that introduced database changes is often not possible. Even when it is, rolling forward is usually a better option when practicing continuous deployment with high-frequency releases limited to a small scope of changes.

We did our best to discourage you from rolling back. Still, in some cases that is a better option. In others, that might be the only option. Luckily, rolling back is reasonably straightforward with Kubernetes.

## Rolling Back with Kubernetes [#](https://www.educative.io/courses/practical-guide-to-kubernetes/x1wL3zynOYE#rolling-back-with-kubernetes)

We’ll imagine that we just discovered that the latest release of the vfarcic/go-demo-2 image is faulty and that we should roll back to the previous release. The command that will do just that is as follows.

kubectl rollout undo \

    -f deploy/go-demo-2-api.yml

kubectl describe \

    -f deploy/go-demo-2-api.yml

The **output** of the latter command, limited to the last lines, is as follows.

OldReplicaSets:  <none>

NewReplicaSet:   go-demo-2-api-68df567fb5 (3/3 replicas created)

Events:

  Type   Reason             Age             From                  Message

  ----   ------             ----            ----                  -------

  Normal ScalingReplicaSet  6m              deployment-controller Scaled up replica set go-demo-2-api-68c75f4f5 to 1

  Normal ScalingReplicaSet  6m              deployment-controller Scaled down replica set go-demo-2-api-68df567fb5 to 2

  Normal ScalingReplicaSet  6m              deployment-controller Scaled up replica set go-demo-2-api-68c75f4f5 to 2

  Normal ScalingReplicaSet  6m              deployment-controller Scaled down replica set go-demo-2-api-68df567fb5 to 1

  Normal ScalingReplicaSet  6m              deployment-controller Scaled up replica set go-demo-2-api-68c75f4f5 to 3

  Normal ScalingReplicaSet  6m              deployment-controller Scaled down replica set go-demo-2-api-68df567fb5 to 0

  Normal DeploymentRollback 1m              deployment-controller Rolled back deployment "go-demo-2-api" to revision 1

  Normal ScalingReplicaSet  1m              deployment-controller Scaled up replica set go-demo-2-api-68df567fb5 to 1

  Normal ScalingReplicaSet  1m              deployment-controller Scaled down replica set go-demo-2-api-68c75f4f5 to 2

  Normal ScalingReplicaSet  1m (x2 over 6m) deployment-controller Scaled up replica set go-demo-2-api-68df567fb5 to 3

  Normal ScalingReplicaSet  1m (x3 over 1m) deployment-controller (combined from similar events): Scaled down replica set go-demo-2-api-68c75f4f5 to0

We can see from the events section that the Deployment initiated rollback and, from there on, the process we experienced before was reversed. It started increasing the replicas of the older ReplicaSet, and decreasing those from the latest one. Once the process is finished, the older ReplicaSet became active with all the replicas, and the newer one was scaled down to zero.

The end result might be easier to see from the NewReplicaSet entry located just above Events. Before we undid the rollout, the value was go-demo-2-api-68c75f4f5, and now it’s go-demo-2-api-68df567fb5.

## Getting the Rollout History [#](https://www.educative.io/courses/practical-guide-to-kubernetes/x1wL3zynOYE#getting-the-rollout-history)

Knowing only the current state of the latest Deployment is often insufficient, and we might need a list of the past rollouts. We can get it with the kubectl rollout history command.

kubectl rollout history \

    -f deploy/go-demo-2-api.yml

The **output** is as follows.

REVISION  CHANGE-CAUSE

2         kubectl set image api=vfarcic/go-demo-2:2.0 --filename=deploy/go-demo-2-api.yml

3         kubectl create --filename=deploy/go-demo-2-api.yml --record=true

If you look at the third revision, you’ll notice that the change cause is the same command we used to create the Deployment the first time. Before we executed kubectl rollout undo, we had two revisions; 1 and 2. The undo command checked the second-to-last revision (1).

Since new deployments do no destroy ReplicaSets but scale them to 0, all it had to do to undo the last change was to scale it back to the desired number of replicas and, at the same time, scale the current one to zero.

# Playing around with the Deployment

In this lesson, we will deploy a few new releases and will play around with the Deployment to explore multiple options.

* [Deploying New Releases](https://www.educative.io/courses/practical-guide-to-kubernetes/BnKX1QmpvvN#deploying-new-releases)
* [Checking the Rollout History](https://www.educative.io/courses/practical-guide-to-kubernetes/BnKX1QmpvvN#checking-the-rollout-history)
* [Undoing the Rollout](https://www.educative.io/courses/practical-guide-to-kubernetes/BnKX1QmpvvN#undoing-the-rollout)

## Deploying New Releases [#](https://www.educative.io/courses/practical-guide-to-kubernetes/BnKX1QmpvvN#deploying-new-releases)

Let’s deploy a few new releases. That will provide us with a broader playground to explore a few additional things we can do with Deployments.

kubectl set image \

    -f deploy/go-demo-2-api.yml \

    api=vfarcic/go-demo-2:3.0 \

    --record

kubectl rollout status \

    -f deploy/go-demo-2-api.yml

We updated the image to vfarcic/go-demo-2:3.0 and retrieved the rollout status. The last line of the **output** of latter command is as follows.

deployment "go-demo-2-api" successfully rolled out

The deployment was successfully updated and, as a result, it created a new ReplicaSet and scaled it up to the desired number of replicas. The previously active ReplicaSet was scaled to 0. As a result, we’re running tag 3.0 of the vfarcic/go-demo-2 image.

We’ll repeat the process with the tag 4.0.

kubectl set image \

-f deploy/go-demo-2-api.yml \

api=vfarcic/go-demo-2:4.0 \

--record

kubectl rollout status \

-f deploy/go-demo-2-api.yml

The **output** of the last line of the rollout status confirmed that the rollout was successful.

## Checking the Rollout History [#](https://www.educative.io/courses/practical-guide-to-kubernetes/BnKX1QmpvvN#checking-the-rollout-history)

Now that we deployed a few releases, we can check the current rollout history.

kubectl rollout history \

-f deploy/go-demo-2-api.yml

The **output** is as follows.

deployments "go-demo-2-api"

REVISION CHANGE-CAUSE

2 kubectl set image api=vfarcic/go-demo-2:2.0 --filename=deploy/go-demo-2-api.yml --record=true

3 kubectl create --filename=deploy/go-demo-2-api.yml --record=true

4 kubectl set image api=vfarcic/go-demo-2:3.0 --filename=deploy/go-demo-2-api.yml --record=true

5 kubectl set image api=vfarcic/go-demo-2:4.0 --filename=deploy/go-demo-2-api.yml --record=true

We can clearly see the commands that produced the changes and, through them, how our application progressed all the way until the current release based on the image vfarcic/go-demo-2:4.0.

You saw that we can rollback to the previous release through the kubectl rollout undo command. In most cases, that should be the correct action when faced with problems and without the ability to roll forward by creating a new release with the fix. However, sometimes even that is not enough, and we have to go back in time further than the previous release.

## Undoing the Rollout [#](https://www.educative.io/courses/practical-guide-to-kubernetes/BnKX1QmpvvN#undoing-the-rollout)

Let’s say that we discovered not only that the current release is faulty but also that a few before it have bugs as well. Following the same narrative, we’ll imagine that the last correct release was based on the image vfarcic/go-demo-2:2.0. We can remedy that by executing the command that follows.

⚠️ Please do NOT run it.

kubectl set image \

-f deploy/go-demo-2-api.yml \

api=vfarcic/go-demo-2:2.0 \

--record

While that command would certainly fix the problem, there is an easier way to accomplish the same result. We can undo the rollout by moving to the last revision that worked correctly. Assuming that we want to revert to the image vfarcic/go-demo-2:2.0, reviewing the change causes listed in the history tells us we should roll back to revision 2. That can be accomplished through the --to-revision argument. The command is as follows.

kubectl rollout undo \

-f deploy/go-demo-2-api.yml \

--to-revision=2

kubectl rollout history \

-f deploy/go-demo-2-api.yml

We undid the rollout by moving to revision 2. We also retrieved the history.

The **output** of the latter command is as follows.

deployments "go-demo-2-api"

REVISION CHANGE-CAUSE

3 kubectl create --filename=deploy/go-demo-2-api.yml --record=true

4 kubectl set image api=vfarcic/go-demo-2:3.0 --filename=deploy/go-demo-2-api.yml --record=true

5 kubectl set image api=vfarcic/go-demo-2:4.0 --filename=deploy/go-demo-2-api.yml --record=true

6 kubectl set image api=vfarcic/go-demo-2:2.0 --filename=deploy/go-demo-2-api.yml --record=true

Through the new revision 6, we can see that the currently active Deployment is based on the image vfarcic/go-demo-2:2.0.

We successfully moved back to a specific point in time. The problem is solved and, if this was the “real” application running in a production cluster, our users would continue interacting with the version of our software that actually works.

# Rolling Back Failed Deployments

In this lesson, we will learn how to roll back the failed deployments.

###### We'll cover the following

* [Why to Roll Back?](https://www.educative.io/courses/practical-guide-to-kubernetes/m2yRPp5MODr#why-to-roll-back)
* [Verification](https://www.educative.io/courses/practical-guide-to-kubernetes/m2yRPp5MODr#verification)
  + [Undo Rollout](https://www.educative.io/courses/practical-guide-to-kubernetes/m2yRPp5MODr#undo-rollout)

## Why to Roll Back? [#](https://www.educative.io/courses/practical-guide-to-kubernetes/m2yRPp5MODr#why-to-roll-back)

Discovering a critical bug is probably the most common reason for a rollback. Still, there are others. For example, we might be in a situation when Pods cannot be created. An easy to reproduce case would be an attempt to deploy an image with a tag that does not exist.

kubectl set image \

   -f deploy/go-demo-2-api.yml \

   api=vfarcic/go-demo-2:does-not-exist \

   --record

The **output** is as follows.

deployment "go-demo-2-api" image updated

After seeing such a message, you might be under the impression that everything is OK. However, that output only indicates that the definition of the image used in the Deployment was successfully updated. That does not mean that the Pods behind the ReplicaSet are indeed running. For one, I can assure you that the vfarcic/go-demo-2:does-not-exist image does not exist.

ℹ️ Please make sure that at least 60 seconds have passed since you executed the kubectl set image command. If you’re wondering why we are waiting, the answer lies in the progressDeadlineSeconds field set in the go-demo-2-api Deployment definition. That’s how much the Deployment has to wait before it deduces that it cannot progress due to a failure to run a Pod.

## Verification [#](https://www.educative.io/courses/practical-guide-to-kubernetes/m2yRPp5MODr#verification)

Let’s take a look at the ReplicaSets.

kubectl get rs -l type=api

The **output** is as follows.

NAME DESIRED CURRENT READY AGE

go-demo-2-api-5b49d94f9b 0 0 0 8m

go-demo-2-api-68c75f4f5 2 2 2 9m

go-demo-2-api-7cb9bb5675 0 0 0 8m

go-demo-2-api-68df567fb5 0 0 0 9m

go-demo-2-api-dc7877dcd 2 2 0 4m

By now, under different circumstances, all the Pods from the new ReplicaSet (go-demo-2-api-dc7877dcd) should be set to 3, and the Pods of the previous one (go-demo-2-api-68c75f4f5) should have been scaled down to 0. However, the Deployment noticed that there is a problem and stopped the update process.

We should be able to get more detailed information with the kubectl rollout status command.

kubectl rollout status -f deploy/go-demo-2-api.yml

The **output** is as follows.

error: deployment "go-demo-2-api" exceeded its progress deadline

The Deployment realized that it shouldn’t proceed. The new Pods are not running, and the limit was reached. There’s no point to continue trying.

If you expected that the Deployment would roll back after it failed, you’re wrong. It will not do such a thing. At least, not without additional addons. That does not mean that we would expect you to sit in front of your terminal, wait for timeouts, and check the rollout status before deciding whether to keep the new update or to roll back. You should deploy new releases as part of your automated CDP pipeline. Fortunately, the status command returns 1 if the deployment failed and we can use that information to decide what to do next. For those of you not living and breathing Linux, any exit code different than 0 is considered an error. Let’s confirm that by checking the exit code of the last command.

echo $?

The **output** is indeed 1, thus confirming that the rollout failed.

We’ll explore automated CDP pipeline soon. For now, just remember that we can find out whether Deployment updates were successful or not.

### Undo Rollout [#](https://www.educative.io/courses/practical-guide-to-kubernetes/m2yRPp5MODr#undo-rollout)

Now that we discovered that our last rollout failed, we should undo it. You already know how to do that, but we’ll remind you just in case you’re of a forgetful nature.

kubectl rollout undo -f deploy/go-demo-2-api.yml

kubectl rollout status -f deploy/go-demo-2-api.yml

The **output** of the last command confirmed that deployment "go-demo-2-api" was successfully rolled out.

Now that we have learned how to rollback no matter whether the problem is a critical bug or inability to run the new release, we can take a short pause from learning new stuff and merge all the definitions we explored thus far into a single YAML file. But, before we do that, we’ll remove the objects we created.

kubectl delete -f deploy/go-demo-2-db.yml

kubectl delete -f deploy/go-demo-2-db-svc.yml

kubectl delete -f deploy/go-demo-2-api.yml

# Merging Everything into the Same YAML Definition

In this lesson, we will merge all the definitions explored until now into a single YAML file and then create objects using that file.

* [Looking into the Merged File](https://www.educative.io/courses/practical-guide-to-kubernetes/N734GokYX6K#looking-into-the-merged-file)
* [The Differences](https://www.educative.io/courses/practical-guide-to-kubernetes/N734GokYX6K#the-differences)
* [Creating Objects with the Merged File](https://www.educative.io/courses/practical-guide-to-kubernetes/N734GokYX6K#creating-objects-with-the-merged-file)

## Looking into the Merged File [#](https://www.educative.io/courses/practical-guide-to-kubernetes/N734GokYX6K#looking-into-the-merged-file)

Consider this lesson a short intermezzo. We’ll merge the definitions we used in this chapter into a single YAML file.

You already had a similar example before, so there’s no need for lengthy explanations.

cat deploy/go-demo-2.yml

The **output** is as follows.

apiVersion: apps/v1

kind: Deployment

metadata:

  name: go-demo-2-db

  labels:

    type: db

    service: go-demo-2

    vendor: MongoLabs

spec:

  selector:

    matchLabels:

      type: db

      service: go-demo-2

  strategy:

    type: Recreate

  template:

    metadata:

      labels:

        type: db

        service: go-demo-2

        vendor: MongoLabs

    spec:

      containers:

      - name: db

        image: mongo:3.3

---

## The Differences [#](https://www.educative.io/courses/practical-guide-to-kubernetes/N734GokYX6K#the-differences)

If you start searching for differences with the previous definitions, you will find a few.

* The minReadySeconds, progressDeadlineSeconds, revisionHistoryLimit, and strategy fields are removed from the go-demo-2-api Deployment.
* We used them mostly as a way to demonstrate their usage. But, since Kubernetes has sensible defaults, we omitted them from this definition.
* You’ll also notice that there are two Services even though we created only one in this chapter. We did not need the go-demo-2-api Service in our examples since we didn’t need to access the API. But, for the sake of completeness, it is included in this definition.
* Finally, the strategy for deploying the database is set to recreate. As explained earlier, it is more suited for a single-replica database, even though we did not mount a volume that would preserve the data.

## Creating Objects with the Merged File [#](https://www.educative.io/courses/practical-guide-to-kubernetes/N734GokYX6K#creating-objects-with-the-merged-file)

Let’s create the objects defined in deploy/go-demo-2.yml. Remember, with --save-config we’re making sure we can edit the configuration later. The alternative would be to use kubectl apply instead.

kubectl create -f deploy/go-demo-2.yml --record --save-config

kubectl get -f deploy/go-demo-2.yml

The **output** of the latter command is as follows.

NAME DESIRED UP-TO-DATE AVAILABLE AGE

deploy/go-demo-2-db 1 1 1 1 15s

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

svc/go-demo-2-db ClusterIP 10.0.0.125 <none> 27017/TCP 15s

NAME DESIRED UP-TO-DATE AVAILABLE AGE

deploy/go-demo-2-api 3 3 3 15s

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

svc/go-demo-2-api NodePort 10.0.0.57 <none> 8080:31586/TCP 15s

All four objects (two Deployments and two Services) were created successfully.

# Updating Multiple Objects

In this lesson, we will learn how to update multiple deployments at a time.

* [The Use of Selector Labels](https://www.educative.io/courses/practical-guide-to-kubernetes/xlXgxZVr6QJ#the-use-of-selector-labels)
* [Defining the Deployment](https://www.educative.io/courses/practical-guide-to-kubernetes/xlXgxZVr6QJ#defining-the-deployment)
* [Creating the Deployment](https://www.educative.io/courses/practical-guide-to-kubernetes/xlXgxZVr6QJ#creating-the-deployment)
* [Updating the Deployments](https://www.educative.io/courses/practical-guide-to-kubernetes/xlXgxZVr6QJ#updating-the-deployments)

## The Use of Selector Labels [#](https://www.educative.io/courses/practical-guide-to-kubernetes/xlXgxZVr6QJ#the-use-of-selector-labels)

Even though most of the time we send requests to specific objects, almost everything is happening using selector labels. When we updated the Deployments, they looked for matching selectors to choose which ReplicaSets to create and scale. They, in turn, created or terminated Pods also using the matching selectors.

Almost everything in Kubernetes is operated using label selectors. It’s just that sometimes that is obscured from us.

We do not have to update an object only by specifying its name or the YAML file where its definition resides. We can also use labels to decide which object should be updated. That opens some interesting possibilities since the selectors might match multiple objects.

## Defining the Deployment [#](https://www.educative.io/courses/practical-guide-to-kubernetes/xlXgxZVr6QJ#defining-the-deployment)

Imagine that we are running several Deployments with Mongo databases and that the time has come to update them all to a newer release. Before we explore how we could do that, we’ll create another Deployment so that we have at least two with the database Pods.

Let us first take a look at the definition.

cat deploy/different-app-db.yml

The **output** is as follows.

apiVersion: apps/v1

kind: Deployment

metadata:

  name: different-app-db

  labels:

    type: db

    service: different-app

    vendor: MongoLabs

spec:

  selector:

    matchLabels:

      type: db

      service: different-app

  template:

    metadata:

      labels:

        type: db

        service: different-app

        vendor: MongoLabs

    spec:

      containers:

      - name: db

        image: mongo:3.3

        ports:

        - containerPort: 27017

When compared with the go-demo-2-db Deployment, the only difference is in the service label. Both have the type set to db.

## Creating the Deployment [#](https://www.educative.io/courses/practical-guide-to-kubernetes/xlXgxZVr6QJ#creating-the-deployment)

Let’s create the deployment.

kubectl create \

    -f deploy/different-app-db.yml

Now that we have two deployments with the mongo:3.3 Pods, we can try to update them both at the same time.

The trick is to find a label (or a set of labels) that uniquely identifies all the Deployments we want to update.

Let’s take a look at the list of Deployments with their labels.

kubectl get deployments --show-labels

The **output** is as follows.

NAME             DESIRED  UP-TO-DATE AVAILABLE AGE LABELS

different-app-db 1        1          1         1h  service=different-app,type=db,vendor=MongoLabs

go-demo-2-api    3        3          3         1h  language=go,service=go-demo-2,type=api

go-demo-2-db     1        1          1         1h  service=go-demo-2,type=db,vendor=MongoLabs

We want to update mongo Pods created using different-app-db and go-demo-2-db Deployments. Both are uniquely identified with the labels type=db and vendor=MongoLabs. Let’s test that.

kubectl get deployments \

    -l type=db,vendor=MongoLabs

The **output** is as follows.

NAME             DESIRED UP-TO-DATE AVAILABLE AGE

different-app-db 1       1          1         1h

go-demo-2-db     1       1          1         1h

## Updating the Deployments [#](https://www.educative.io/courses/practical-guide-to-kubernetes/xlXgxZVr6QJ#updating-the-deployments)

We can see that filtering with those two labels worked. We retrieved only the Deployments we want to update, so let’s proceed and roll out the new release.

kubectl set image deployments \

    -l type=db,vendor=MongoLabs \

    db=mongo:3.4 --record

The **output** is as follows.

deployment.extensions/different-app-db image updated

deployment.extensions/go-demo-2-db image updated

Finally, before we move into the next subject, we should validate that the image indeed changed to mongo:3.4.

kubectl describe \

    -f deploy/go-demo-2.yml

The **output**, limited to the relevant parts, is as follows.

...

  Containers:

   db:

    Image:        mongo:3.4

...

As we can see, the update was indeed successful, at least with that Deployment. Feel free to describe the Deployment defined in deploy/different-app-db.yml. You should see that its image was also updated to the newer version.

# Scaling Deployments

In this lesson, we will learn to scale deployments using a YAML file and will discuss automated scaling briefly.

* [Scaling Using YAML Files](https://www.educative.io/courses/practical-guide-to-kubernetes/B6vxyNv6NlJ#scaling-using-yaml-files)
* [Looking into the File](https://www.educative.io/courses/practical-guide-to-kubernetes/B6vxyNv6NlJ#looking-into-the-file)
* [Applying the Definition](https://www.educative.io/courses/practical-guide-to-kubernetes/B6vxyNv6NlJ#applying-the-definition)
  + [Verification](https://www.educative.io/courses/practical-guide-to-kubernetes/B6vxyNv6NlJ#verification)
* [Automated Scaling](https://www.educative.io/courses/practical-guide-to-kubernetes/B6vxyNv6NlJ#automated-scaling)
* [Scaling the Deployment](https://www.educative.io/courses/practical-guide-to-kubernetes/B6vxyNv6NlJ#scaling-the-deployment)
  + [Verification](https://www.educative.io/courses/practical-guide-to-kubernetes/B6vxyNv6NlJ#verification-2)

## Scaling Using YAML Files [#](https://www.educative.io/courses/practical-guide-to-kubernetes/B6vxyNv6NlJ#scaling-using-yaml-files)

There are quite a few different ways we can scale Deployments. Everything we do in this section is not unique to Deployments and can be applied to any Controller, like ReplicaSet, and those we did not yet explore.

If we decide that the number of replicas changes with relatively low frequency or that Deployments are performed manually, the best way to scale is to write a new YAML file or, even better, modify the existing one. Assuming that we store YAML files in a code repository, by updating existing files we have a documented and reproducible definition of the objects running inside a cluster.

We already performed scaling when we applied the definition from the go-demo-2-scaled.yml. We’ll do something similar, but with Deployments.

## Looking into the File [#](https://www.educative.io/courses/practical-guide-to-kubernetes/B6vxyNv6NlJ#looking-into-the-file)

Let’s take a look at deploy/go-demo-2-scaled.yml.

cat deploy/go-demo-2-scaled.yml

We won’t display the contents of the whole file since it is almost identical deploy/go-demo-2.yml. The only difference is the number of replicas of the go-demo-2-api Deployment.

...

apiVersion: apps/v1

kind: Deployment

metadata:

  name: go-demo-2-api

spec:

  replicas: 5

...

## Applying the Definition [#](https://www.educative.io/courses/practical-guide-to-kubernetes/B6vxyNv6NlJ#applying-the-definition)

At the moment, we’re running three replicas. Once we apply the new definition, it should increase to five.

kubectl apply \

    -f deploy/go-demo-2-scaled.yml

Please note that, even though the file is different, the names of the resources are the same so kubectl apply did not create new objects. Instead, it updated those that changed. In particular, it changed the number of replicas of the go-demo-2-api Deployment.

### Verification [#](https://www.educative.io/courses/practical-guide-to-kubernetes/B6vxyNv6NlJ#verification)

Let’s confirm that there are indeed five replicas of the Pods controlled through the Deployment.

kubectl get \

    -f deploy/go-demo-2-scaled.yml

The **output**, limited to the deploy/go-demo-2-api, is as follows.

...

NAME                            READY   UP-TO-DATE   AVAILABLE   AGE

deployment.apps/go-demo-2-api   5/5     5            5           3h26m

...

The result should come as no surprise. After all, we executed the same process before, when we explored ReplicaSets.

## Automated Scaling [#](https://www.educative.io/courses/practical-guide-to-kubernetes/B6vxyNv6NlJ#automated-scaling)

While scaling Deployments using YAML files (or other Controllers) is an excellent way to keep documentation accurate, it rarely fits the dynamic nature of the clusters. We should aim for a system that will scale (and de-scale) services automatically.

When scaling is frequent and, hopefully, automated, we cannot expect to update YAML definitions and push them to Git. That would be too inefficient and would probably cause quite a few unwanted executions of delivery pipelines if they are triggered through repository WebHooks. After all, do we really want to push updated YAML files multiple times a day?

The number of replicas should not be part of the design. Instead, they are a fluctuating number that changes continuously (or at least often), depending on the traffic, memory and CPU utilization, and so on.

Depending on release frequency, the same can be said for image. If we are practicing continuous delivery or deployment, we might be releasing once a week, once a day, or even more often. In such cases, new images would be deployed often, and there is no strong argument for the need to change YAML files every time we make a new release. That is especially true if we are deploying through an automated process (as we should).

We’ll explore automation later on. For now, we’ll limit ourselves to a command similar to kubectl set image. We used it to change the image used by Pods with each release.

## Scaling the Deployment [#](https://www.educative.io/courses/practical-guide-to-kubernetes/B6vxyNv6NlJ#scaling-the-deployment)

Similarly, we’ll use kubectl scale to change the number of replicas. Consider this an introduction to automation that is coming later on.

kubectl scale deployment \

    go-demo-2-api --replicas 8 --record

We scaled the number of replicas associated with the Deployment go-demo-2-api. Please note that, this time, we did not use -f to reference a file. Since we have two Deployments specified in the same YAML, that would result in scaling of both. Since we wanted to limit it to a particular Deployment, we used its name instead.

### Verification [#](https://www.educative.io/courses/practical-guide-to-kubernetes/B6vxyNv6NlJ#verification-2)

Let’s confirm that scaling indeed worked as expected.

kubectl get -f deploy/go-demo-2.yml

The **output**, limited to Deployments, is as follows.

NAME                           READY   UP-TO-DATE   AVAILABLE   AGE

deployment.apps/go-demo-2-db   1/1     1            1           4h40m

NAME                            READY   UP-TO-DATE   AVAILABLE   AGE

deployment.apps/go-demo-2-api   8/8     8            8           3h28m

As we mentioned earlier, we’ll dedicate quite a lot of time to automation, and you won’t have to scale your applications manually. However, it is useful to know that the kubectl scale command exists. For now, you know how to scale Deployments (and other Controllers).

Diagram

Description automatically generated

# Getting Started with Ingress

In this lesson, we will learn what is ingress and why it should be used.

* [Why Use Ingress Objects?](https://www.educative.io/courses/practical-guide-to-kubernetes/gx5BpRMN6A6#why-use-ingress-objects)
* [Creating A Cluster](https://www.educative.io/courses/practical-guide-to-kubernetes/gx5BpRMN6A6#creating-a-cluster)

## Why Use Ingress Objects? [#](https://www.educative.io/courses/practical-guide-to-kubernetes/gx5BpRMN6A6#why-use-ingress-objects)

Applications that are not accessible to users are useless. Kubernetes Services provide accessibility with a usability cost. Each application can be reached through a different port. We **cannot** expect users to know the port of each service in our cluster.

Ingress objects manage external access to the applications running inside a Kubernetes cluster.

While, at first glance, it might seem that we already accomplished that through Kubernetes Services, they do not make the applications truly accessible. We still need forwarding rules based on paths and domains, SSL termination and a number of other features.

In a more traditional setup, we’d probably use an external proxy and a load balancer. Ingress provides an API that allows us to accomplish these things, in addition to a few other features we expect from a dynamic cluster.

We’ll explore the problems and the solutions through examples. For now, we need first to create a cluster.

ℹ️ All the commands from this chapter are available in the [07-ingress.sh](https://gist.github.com/54ef6592bce747ff2d1b089834fc755b) Gist.

## Creating A Cluster [#](https://www.educative.io/courses/practical-guide-to-kubernetes/gx5BpRMN6A6#creating-a-cluster)

As every other chapter so far, we’ll start by creating a Minikube single-node cluster.

cd k8s-specs

git pull

minikube start --vm-driver=virtualbox

kubectl config current-context

The cluster should be up-and-running, and we can move on.

# Why Services Are Not the Best Fit for External Access?

In this lesson, we will discover why services are not the best fit for enabling external access to the applications.

* [Only Services won’t Suffice](#Only Services won’t Suffice )
* [Access Through Services](#Access Through Services )
* [Understanding the Process](#Understanding the Process )
* [The Solution](#The Solution )
* [SSL Certificates](#SSL Certificates )

## Only Services won’t Suffice [#](https://www.educative.io/courses/practical-guide-to-kubernetes/7ngo8gywQmr#only-services-wont-suffice)

We cannot explore solutions before we know what the problems are. Therefore, we’ll re-create a few objects using the knowledge we already gained. That will let us see whether Kubernetes Services satisfy all the needs users of our applications might have. Or, to be more explicit, we’ll explore which features we’re missing when making our applications accessible to users.

We already discussed that it is a bad practice to publish fixed ports through Services. That method is likely to result in conflicts or, at the very least, create the additional burden of carefully keeping track of which port belongs to which Service. We already discarded that option before, and we won’t change our minds now.

Since we’ve clarified that, let’s go back and create the Deployments and the Services from the previous chapter.

kubectl create \

    -f ingress/go-demo-2-deploy.yml

kubectl get \

    -f ingress/go-demo-2-deploy.yml

The **output** of the get command is as follows.

NAME                           READY   UP-TO-DATE   AVAILABLE   AGE

deployment.apps/go-demo-2-db   0/1     1            0           2m15s

NAME                   TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)     AGE

service/go-demo-2-db   ClusterIP   10.111.211.179   <none>        27017/TCP   2m15s

NAME                            READY   UP-TO-DATE   AVAILABLE   AGE

deployment.apps/go-demo-2-api   0/3     3            0           2m15s

NAME                    TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE

service/go-demo-2-api   NodePort   10.103.180.226   <none>        8080:30753/TCP   2m15s

As you can see, these are the same Services and Deployments we previously created.

Before we move on, we should wait until all the Pods are up and running.

kubectl get pods

The **output** is as follows.

NAME                           READY STATUS  RESTARTS AGE

go-demo-2-api-68df567fb5-8qcmv 1/1   Running 0        3m

go-demo-2-api-68df567fb5-k55d4 1/1   Running 0        3m

go-demo-2-api-68df567fb5-ws9cj 1/1   Running 0        3m

go-demo-2-db-dd48b7dfc-hdxbz   1/1   Running 0        3m

If, in your case, some of the Pods are not yet running, please wait a few moments and re-execute the kubectl get pods command. We’ll continue once they’re ready.

## Access Through Services [#](https://www.educative.io/courses/practical-guide-to-kubernetes/7ngo8gywQmr#access-through-services)

One obvious way to access the applications is through Services.

IP=$(minikube ip)

PORT=$(kubectl get svc go-demo-2-api \

    -o jsonpath="{.spec.ports[0].nodePort}")

curl -i "http://$IP:$PORT/demo/hello"

We retrieved the Minikube IP and the port of the go-demo-2-api Service. We used that information to send a request.

The **output** of the curl command is as follows.

HTTP/1.1 200 OK

Date: Sun, 24 Dec 2017 13:35:26 GMT

Content-Length: 14

Content-Type: text/plain; charset=utf-8

hello, world!

The application responded with the status code 200 thus confirming that the Service indeed forwards the requests.

While publishing a random, or even a hard-coded port of a single application might not be so bad, if we’d apply the same principle to more applications, the user experience would be horrible. To make the point a bit clearer, we’ll deploy another application.

kubectl create \

    -f ingress/devops-toolkit-dep.yml \

    --record --save-config

kubectl get \

    -f ingress/devops-toolkit-dep.yml

This application follows similar logic to the first. From the latter command, we can see that it contains a Deployment and a Service. The details are of no importance since the YAML definition is very similar to those we used before. What matters is that now we have two applications running inside the cluster.

## Understanding the Process [#](https://www.educative.io/courses/practical-guide-to-kubernetes/7ngo8gywQmr#understanding-the-process)

Let’s check whether the new application is indeed reachable.

PORT=$(kubectl get svc devops-toolkit \

    -o jsonpath="{.spec.ports[0].nodePort}")

open "http://$IP:$PORT"

We retrieved the port of the new Service and opened the application in a browser. If you get a page not found error, you might want to wait a bit longer until the containers are pulled, and try again

A simplified flow of requests is depicted in the below-given illustration.

A user sends a request to one of the nodes of the cluster. That request is received by a Service and load balanced to one of the associated Pods. It’s a bit more complicated than that, with iptables, kube DNS, kube proxy, and a few other things involved in the process. We explored them in more detail in the Using Services To Enable Communication Between Pods chapter, and there’s probably no need to go through them all again. For the sake of brevity, the simplified diagram should do.

Diagram

Description automatically generated

Applications access through Services

We cannot expect our users to know specific ports behind each of those applications. Even with only two, that would not be very user-friendly. If that number would rise to tens or even hundreds of applications, our business would be very short-lived.

What we need is a way to make all services accessible through standard HTTP (80) or HTTPS (443) ports. Kubernetes Services alone cannot get us there. We need more.

## The Solution [#](https://www.educative.io/courses/practical-guide-to-kubernetes/7ngo8gywQmr#the-solution)

What we need is to grant access to our services on predefined paths and domains. Our go-demo-2 service could be distinguished from others through the base path /demo. Similarly, the books application could be reachable through the devopstoolkitseries.com domain. If we could accomplish that, we could access them with the commands as follows.

curl "http://$IP/demo/hello"

The request received the default backend - 404 response. There is no process listening on port 80, so this outcome is not a surprise. We could have changed one of the Services to publish the fixed port 80 instead assigning a random one. Still, that would provide access only to one of the two applications.

We often want to associate each application with a different domain or sub-domain. Outside the examples we’re running, the books application is accessible through the [devopstoolkitseries.com](http://www.devopstoolkitseries.com/) domain. Since access to the domain is not feasible, we’ll simulate it by adding the domain to the Host header.

The command that should verify whether the application running inside our cluster is accessible through the devopstoolkitseries.com domain is as follows.

curl -i \

    -H "Host: devopstoolkitseries.com" \

    "http://$IP"

As expected, the request is still refused.

## SSL Certificates [#](https://www.educative.io/courses/practical-guide-to-kubernetes/7ngo8gywQmr#ssl-certificates)

Last, but not the least, we should be able to make some, if not all, applications (partly) secure by enabling HTTPS access. That means that we should have a place to store our SSL certificates. We could put them inside our applications, but that would only increase the operational complexity. Instead, we should aim towards SSL offloading somewhere between clients and the applications, and it should come as no surprise that Kubernetes has a solution for all these.

# Getting Started with Volumes

In this lesson, you will be introduced to Kubernetes Volumes and create a Minikube cluster.

###### We'll cover the following

* [State Preservation](https://www.educative.io/courses/practical-guide-to-kubernetes/JQBRwXwLkLv#state-preservation)
* [The Volumes](https://www.educative.io/courses/practical-guide-to-kubernetes/JQBRwXwLkLv#the-volumes)
* [Copying the Files](https://www.educative.io/courses/practical-guide-to-kubernetes/JQBRwXwLkLv#copying-the-files)
* [Creating a Cluster](https://www.educative.io/courses/practical-guide-to-kubernetes/JQBRwXwLkLv#creating-a-cluster)

## State Preservation [#](https://www.educative.io/courses/practical-guide-to-kubernetes/JQBRwXwLkLv#state-preservation)

Having a system without a state is impossible. Even though there is a tendency to develop stateless applications, we still need to deal with the state. There are databases and other stateful third-party applications. No matter what we do, we need to make sure that the state is preserved no matter what happens to containers, Pods, or even whole nodes.

Most of the time, stateful applications store their state on disk. That leaves us with a problem. If a container crashes, kubelet will restart it. The problem is that it will create a new container based on the same image. All data accumulated inside a container that crashed will be lost.

## The Volumes [#](https://www.educative.io/courses/practical-guide-to-kubernetes/JQBRwXwLkLv#the-volumes)

**Kubernetes Volumes** solve the need to preserve the state across container crashes. In essence, Volumes are references to files and directories made accessible to containers that form a Pod. The significant difference between different types of Kubernetes Volumes is in the way these files and directories are created.

While the primary use-case for Volumes is the preservation of state, there are quite a few others. For example, we might use Volumes to access Docker’s socket running on a host. Or we might use them to access configuration residing in a file on the host file system.

We can describe Volumes as a way to access a file system that might be running on the same host or somewhere else. No matter where that file system is, it is external to the containers that mount volumes. There can be many reasons why someone might mount a Volume, with state preservation being only one of them.

There are over **twenty-five** Volume types supported by Kubernetes. It would take us too much time to go through all of them. Besides, even if we’d like to do that, many Volume types are specific to a hosting vendor. For example, awsElasticBlockStore works only with AWS, azureDisk and azureFile work only with Azure, and so on and so forth.

We’ll limit our exploration to Volume types that can be used within Minikube. You should be able to extrapolate that knowledge to Volume types applicable to your hosting vendor of choice.

Let’s get down to it.

ℹ️ All the commands from this chapter are available in the [08-volume.sh](https://gist.github.com/5acafb64c0124a1965f6d371dd0dedd1) Gist.

## Copying the Files [#](https://www.educative.io/courses/practical-guide-to-kubernetes/JQBRwXwLkLv#copying-the-files)

This time, we’ll have an additional action we’ll execute in preparation to create a Minikube cluster.

cd k8s-specs

git pull

cp volume/prometheus-conf.yml \

    ~/.minikube/files

We’ll need the volume/prometheus-conf.yml file inside the soon-to-be-created Minikube VM. When it starts, it will copy all the files from ~/.minikube/files on your host, into the /files directory in the VM.

⚠️ Depending on your operating system, the ~/.minikube/files directory might be somewhere else. If that’s the case, please adapt the command above.

## Creating a Cluster [#](https://www.educative.io/courses/practical-guide-to-kubernetes/JQBRwXwLkLv#creating-a-cluster)

Now that the files are copied to the shared directory, we can repeat the same process we did quite a few times before.

📝 Please note that we’ve added the step from the last chapter that enables the ingress addon.

minikube start --vm-driver=virtualbox

minikube addons enable ingress

kubectl config current-context

Our Minikube cluster is up and running.

# Accessing Host’s Resources through hostPath Volumes

In this lesson, we will go through the hostPath Volume type and try to access the host's resources through it.

We'll cover the following

* [Building Docker Images](https://www.educative.io/courses/practical-guide-to-kubernetes/3YE001Qky9R#building-docker-images)
  + [Creating a Pod with Docker Image](https://www.educative.io/courses/practical-guide-to-kubernetes/3YE001Qky9R#creating-a-pod-with-docker-image)
  + [Creating a Pod with hostPath](https://www.educative.io/courses/practical-guide-to-kubernetes/3YE001Qky9R#creating-a-pod-with-hostpath)
    - [Looking into the Definition](https://www.educative.io/courses/practical-guide-to-kubernetes/3YE001Qky9R#looking-into-the-definition)
* [The hostPath Volume](https://www.educative.io/courses/practical-guide-to-kubernetes/3YE001Qky9R#the-hostpath-volume)
  + [Types of Mounts in hostPath](https://www.educative.io/courses/practical-guide-to-kubernetes/3YE001Qky9R#types-of-mounts-in-hostpath)

## Building Docker Images [#](https://www.educative.io/courses/practical-guide-to-kubernetes/3YE001Qky9R#building-docker-images)

Sooner or later, we’ll have to build our images. A simple solution would be to execute the docker image build command directly from a server. However, that might cause problems. Building images on a single host means that there is an uneven resource utilization and that there is a single point of failure. Wouldn’t it be better if we could build images anywhere inside a Kubernetes cluster?

Instead of executing the docker image build command, we could create a Pod based on the docker image. Kubernetes will make sure that the Pod is scheduled somewhere inside the cluster, thus distributing resource usage much better.

### Creating a Pod with Docker Image [#](https://www.educative.io/courses/practical-guide-to-kubernetes/3YE001Qky9R#creating-a-pod-with-docker-image)

Let’s start with an elementary example. If we can list the images, we’ll prove that running docker commands inside containers works. Since, from Kubernetes’ point of view, Pods are the smallest entity, that’s what we’ll run.

kubectl run docker \

--image=docker:17.11 \

--generator "run-pod/v1" \

docker image ls

kubectl get pods

We created a Pod named docker and based it on the official docker image. Since we want to execute a one-shot command, we specified that it should Never restart. Finally, the container command is docker image ls. The second command lists all the Pods in the cluster (including failed ones).

The **output** of the latter command is as follows.

NAME READY STATUS RESTARTS AGE

docker 0/1 Error 0 1m

The output should show that the status is Error, thus indicating that there is a problem with the container we’re running. If, in your case, the status is not yet Error, Kubernetes is probably still pulling the image. In that case, please wait a few moments, and re-execute the kubectl get pods command.

Let’s take a look at the logs of the container.

kubectl logs docker

The **output** is as follows.

Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running?

Docker consists of two main pieces. There is a client, and there is a server. When we executed docker image ls, we invoked the client which tried to communicate with the server through its API. The problem is that Docker server is not running in that container. What we should do is tell the client (inside a container) to use Docker server that is already running on the host (Minikube VM).

By default, the client sends instructions to the server through the socket located in /var/run/docker.sock. We can accomplish our goal if we mount that file from the host into a container.

Before we try to enable communication between a Docker client in a container and Docker server on a host, we’ll delete the Pod we created a few moments ago.

kubectl delete pod docker

### Creating a Pod with hostPath [#](https://www.educative.io/courses/practical-guide-to-kubernetes/3YE001Qky9R#creating-a-pod-with-hostpath)

Let’s mount the file /var/run/docker.sock from the host in our Pod.

#### Looking into the Definition [**#**](https://www.educative.io/courses/practical-guide-to-kubernetes/3YE001Qky9R#looking-into-the-definition)

Let’s take a look at the Pod definition stored in volume/docker.yml.

cat volume/docker.yml

The **output** is as follows.

apiVersion: v1

kind: Pod

metadata:

name: docker

spec:

containers:

- name: docker

image: docker:17.11

command: ["sleep"]

args: ["100000"]

volumeMounts:

- mountPath: /var/run/docker.sock

name: docker-socket

volumes:

- name: docker-socket

hostPath:

path: /var/run/docker.sock

type: Socket

Part of the definition closely mimics the kubectl run command we executed earlier. The only significant difference is in the volumeMounts and volumes sections.

**Line 9-10:** We changed the command and the arguments to sleep 100000. That will give us more freedom since we’ll be able to create the Pod, enter inside its only container, and experiment with different commands.

**Line 11:** The volumeMounts field is relatively straightforward and is the same no matter which type of Volume we’re using. In this section, we’re specifying the mountPath and the name of the volume. The former is the path we expect to mount inside this container. You’ll notice that we are not specifying the type of the volume nor any other specifics inside the VolumeMounts section. Instead, we simply have a reference to a volume called docker-socket.

**Line 14:** The Volume configuration specific to each type is defined in the volumes section. In this case, we’re using the hostPath Volume type.

## The hostPath Volume [#](https://www.educative.io/courses/practical-guide-to-kubernetes/3YE001Qky9R#the-hostpath-volume)

hostPath allows us to mount a file or a directory from a host to Pods and, through them, to containers. Before we discuss the usefulness of this type, we’ll have a short discussion about use-cases when this is not a good choice.

**Do not** use hostPath to store a state of an application. Since it mounts a file or a directory from a host into a Pod, it is not fault-tolerant. If the server fails, Kubernetes will schedule the Pod to a healthy node, and the state will be lost.

For our use case, hostPath works just fine. We’re not using it to preserve state, but to gain access to Docker server running on the same host as the Pod.

**Line 15-18:** The hostPath type has only **two** fields. The path represents the file or a directory we want to mount from the host. Since we want to mount a socket, we set the type accordingly. There are other types we could use.

### Types of Mounts in hostPath [#](https://www.educative.io/courses/practical-guide-to-kubernetes/3YE001Qky9R#types-of-mounts-in-hostpath)

* The Directory type will mount a directory from the host. It must exist on the given path. If it doesn’t, we might switch to DirectoryOrCreate type which serves the same purpose. The difference is that DirectoryOrCreate will create the directory if it does not exist on the host.
* The File and FileOrCreate are similar to their Directory equivalents. The only difference is that this time we’d mount a file, instead of a directory.
* The other supported types are Socket, CharDevice, and BlockDevice. They should be self-explanatory. If you don’t know what character or block devices are, you probably don’t need those types.

These were the types of mounts supported by the hostPath.

# Running the Pod after mounting hostPath

In this lesson, we will create the Pod by mounting Docker socket and play around in it.

###### We'll cover the following

* [Creating and Testing the Pod](https://www.educative.io/courses/practical-guide-to-kubernetes/7AYRLMPvwY1#creating-and-testing-the-pod)
* [Playing Around with Docker](https://www.educative.io/courses/practical-guide-to-kubernetes/7AYRLMPvwY1#playing-around-with-docker)
* [Destroying the Pod](https://www.educative.io/courses/practical-guide-to-kubernetes/7AYRLMPvwY1#destroying-the-pod)

## Creating and Testing the Pod [#](https://www.educative.io/courses/practical-guide-to-kubernetes/7AYRLMPvwY1#creating-and-testing-the-pod)

Let’s create the Pod and check whether, this time, we can execute Docker commands from inside the container it’ll create.

kubectl create \

  -f volume/docker.yml

Since the image is already pulled, starting the Pod should be almost instant.

Let’s see whether we can retrieve the list of Docker images.

kubectl exec -it docker \

    -- docker image ls \

    --format "{{.Repository}}"

We executed docker image ls command and shortened the output by limiting its formatting only to Repository. The output is as follows.

k8s.gcr.io/kube-proxy

k8s.gcr.io/kube-controller-manager

k8s.gcr.io/kube-scheduler

k8s.gcr.io/kube-apiserver

quay.io/kubernetes-ingress-controller/nginx-ingress-controller

k8s.gcr.io/kube-addon-manager

k8s.gcr.io/coredns

k8s.gcr.io/kubernetes-dashboard-amd64

k8s.gcr.io/etcd

k8s.gcr.io/k8s-dns-sidecar-amd64

k8s.gcr.io/k8s-dns-kube-dns-amd64

k8s.gcr.io/k8s-dns-dnsmasq-nanny-amd64

k8s.gcr.io/pause

docker

gcr.io/k8s-minikube/storage-provisioner

gcr.io/google\_containers/defaultbackend

Even though we executed the docker command inside a container, the output clearly shows the images from the host. We proved that mounting the Docker socket (/var/run/docker.sock) as a Volume allows communication between Docker client inside the container, and Docker server running on the host.

Diagram

Description automatically generated

## Playing Around with Docker [#](https://www.educative.io/courses/practical-guide-to-kubernetes/7AYRLMPvwY1#playing-around-with-docker)

Let’s enter the container and see whether we can build a Docker image.

kubectl exec -it docker sh

To build an image, we need a Dockerfile as well as an application’s source code. We’ll continue using go-demo-2 as the example, so our first action will be to clone the repository.

apk add -U git

git clone \

    https://github.com/vfarcic/go-demo-2.git

cd go-demo-2

We used apk add to install git.On the other hand, docker and many other images use alpine as the base. If you’re not familiar with alpine, it is a very slim and efficient base image, and we strongly recommend that you use it when building your own.

Images like debian, centos, ubuntu, redhat, and similar base images are often a terrible choice made because of a misunderstanding of how containers work.

alpine uses apk package management, so we invoked it to install git. Next, we cloned the vfarcic/go-demo-2 repository, and, finally, we entered into the go-demo-2 directory.

Let’s take a quick look at the Dockerfile.

cat Dockerfile

The **output** is as follows.

FROM golang:1.9 AS build

ADD . /src

WORKDIR /src

RUN go get -d -v -t

RUN go test --cover -v ./... --run UnitTest

RUN go build -v -o go-demo

FROM alpine:3.4

MAINTAINER      Viktor Farcic <viktor@farcic.com>

RUN mkdir /lib64 && ln -s /lib/libc.musl-x86\_64.so.1 /lib64/ld-linux-x86-64.so.2

EXPOSE 8080

ENV DB db

CMD ["go-demo"]

HEALTHCHECK --interval=10s CMD wget -qO- localhost:8080/demo/hello

COPY --from=build /src/go-demo /usr/local/bin/go-demo

RUN chmod +x /usr/local/bin/go-demo

Since this course is dedicated to Kubernetes, we won’t go into details behind this Dockerfile, but only comment that it uses Docker’s multi-stage builds. The first stage downloads the dependencies, it runs unit tests, and it builds the binary. The second stage starts over. It builds a fresh image with the go-demo binary copied from the previous stage.

ℹ️ We hope you’re proficient with Docker and there’s no need to explain image building further.

Let’s test whether building an image indeed works.

docker image build \

    -t vfarcic/go-demo-2:beta .

docker image ls \

    --format "{{.Repository}}"

We executed the docker image build command, followed by docker image ls. The **output** of the latter command is as follows.

vfarcic/go-demo-2

<none>

golang

docker

alpine

gcr.io/google\_containers/nginx-ingress-controller

gcr.io/google\_containers/k8s-dns-sidecar-amd64

gcr.io/google\_containers/k8s-dns-kube-dns-amd64

gcr.io/google\_containers/k8s-dns-dnsmasq-nanny-amd64

gcr.io/google\_containers/kubernetes-dashboard-amd64

gcr.io/google\_containers/kubernetes-dashboard-amd64

gcr.io/google-containers/kube-addon-manager

gcr.io/google\_containers/defaultbackend

gcr.io/google\_containers/pause-amd64

If we compare this with the previous docker image ls output, we’ll notice that, this time, a few new images are listed. The golang and alpine images are used as a basis for each of the build stages. The vfarcic/go-demo-2 is the result of our build. Finally, <none> is only a left-over of the process and it can be safely removed.

docker system prune -f

docker image ls \

    --format "{{.Repository}}"

The docker system prune command removes all unused resources. At least, all those created and unused by Docker. We confirmed that by executing docker image ls again. This time, we can see the <none> image is gone.

## Destroying the Pod [#](https://www.educative.io/courses/practical-guide-to-kubernetes/7AYRLMPvwY1#destroying-the-pod)

We’ll destroy the docker Pod and explore other usages of the hostPath Volume type.

exit

kubectl delete \

    -f volume/docker.yml

hostPath is a great solution for accessing host resources like /var/run/docker.sock, /dev/cgroups, and others. That is, as long as the resource we’re trying to reach is on the same node as the Pod.

Let’s see whether we can find other use-cases for hostPath.

# Using hostPath Volume Type to Inject Configuration Files

In this lesson, we will get familiar with Prometheus and configure it with hostPath Volume.

* [Using Prometheus](https://www.educative.io/courses/practical-guide-to-kubernetes/3jz9Mg2B8Y4#using-prometheus)
  + [Looking into the Definition](https://www.educative.io/courses/practical-guide-to-kubernetes/3jz9Mg2B8Y4#looking-into-the-definition)
  + [Configuring the IP](https://www.educative.io/courses/practical-guide-to-kubernetes/3jz9Mg2B8Y4#configuring-the-ip)
  + [Testing Prometheus](https://www.educative.io/courses/practical-guide-to-kubernetes/3jz9Mg2B8Y4#testing-prometheus)
  + [Changing Prometheus Configuration](https://www.educative.io/courses/practical-guide-to-kubernetes/3jz9Mg2B8Y4#changing-prometheus-configuration)

## Using Prometheus [#](https://www.educative.io/courses/practical-guide-to-kubernetes/3jz9Mg2B8Y4#using-prometheus)

We are about to deploy [Prometheus](https://prometheus.io/) for the first time. We won’t go into details behind the application except to say that it’s fantastic and that you should consider it for your monitoring and alerting needs. We’re using it only to demonstrate a few Kubernetes concepts. We’re not trying to learn how to operate it.

### Looking into the Definition [#](https://www.educative.io/courses/practical-guide-to-kubernetes/3jz9Mg2B8Y4#looking-into-the-definition)

Let’s take a look at the application’s definition.

cat volume/prometheus.yml

The **output** is as follows.

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

  name: prometheus

  annotations:

    kubernetes.io/ingress.class: "nginx"

    ingress.kubernetes.io/ssl-redirect: "false"

    nginx.ingress.kubernetes.io/ssl-redirect: "false"

spec:

  rules:

  - http:

      paths:

      - path: /prometheus

        backend:

          serviceName: prometheus

          servicePort: 9090

---

apiVersion: apps/v1

kind: Deployment

metadata:

  name: prometheus

spec:

  selector:

    matchLabels:

      type: monitor

      service: prometheus

There’s nothing genuinely new in that YAML file. It defines an Ingress, a Deployment, and a Service. There is, however, one thing we might need to change.

### Configuring the IP [#](https://www.educative.io/courses/practical-guide-to-kubernetes/3jz9Mg2B8Y4#configuring-the-ip)

Prometheus needs a full external-url if we want to change the base path. At the moment, it’s set to the IP of our Minikube VM. In your case, that IP might be different. We’ll fix that by adding a bit of sed “magic” that will make sure the IP matches that of your Minikube VM.

cat volume/prometheus.yml | sed -e \

    "s/192.168.99.100/$(minikube ip)/g" \

    | kubectl create -f - \

    --record --save-config

kubectl rollout status deploy prometheus

We output the contents of the volume/prometheus.yml file, we used sed to replace the hard-coded IP with the actual value of your Minikube instance, and we passed the result to kubectl create.

📝 Please note that, this time, the create command has dash (-) instead of the path to the file. That’s an indication that stdin should be used instead.

Once we created the application, we used the kubectl rollout status command to confirm that the deployment finished.

### Testing Prometheus [#](https://www.educative.io/courses/practical-guide-to-kubernetes/3jz9Mg2B8Y4#testing-prometheus)

Now we can open Prometheus in a browser.

open "http://$(minikube ip)/prometheus"

At first glance, the application seems to be running correctly. However, since the targets are the crucial part of the application, we should check them as well. For those not familiar with Prometheus, it pulls data from targets (external data sources) and, by default, comes with only one target pre-configured: Prometheus itself. Prometheus will always pull data from this target unless we configure it otherwise.

Let’s take a look at its targets.

open "http://$(minikube ip)/prometheus/targets"

There’s something wrong. The default target is not reachable. Before we start panicking, we should take a closer look at its configuration.

open "http://$(minikube ip)/prometheus/config"

The problem is with the metrics\_path field. By default, it is set to /metrics. However, since we changed the base path to /prometheus, the field should have /prometheus/metrics as the value.

### Changing Prometheus Configuration [#](https://www.educative.io/courses/practical-guide-to-kubernetes/3jz9Mg2B8Y4#changing-prometheus-configuration)

Long story short, we must change the Prometheus configuration.

We could, for example, enter the container, update the configuration file, and send the reload request to Prometheus. That would be a terrible solution since it would last only until the next time we update the application, or until the container fails, and Kubernetes decides to reschedule it.

Let’s explore alternative solutions. We could, for example, use hostPath Volume for this as well. If we can guarantee that the correct configuration file is inside the VM, the Pod could attach it to the prometheus container. Let’s try it out.

cat volume/prometheus-host-path.yml

The **output**, limited to relevant parts, is as follows.

apiVersion: apps/v1

kind: Deployment

metadata:

  name: prometheus

spec:

  selector:

    ...

    spec:

      containers:

        ...

        volumeMounts:

        - mountPath: /etc/prometheus/prometheus.yml

          name: prom-conf

      volumes:

      - name: prom-conf

        hostPath:

          path: /files/prometheus-conf.yml

          type: File

...

The only significant difference, when compared with the previous definition, is in the added volumeMounts and volumesfields. We’re using the same schema as before, except that, this time, the type is set to File. Once we apply this Deployment, the file /files/prometheus-conf.yml on the host will be available as /etc/prometheus/prometheus.yml inside the container.

If you recall, we copied one file to the ~/.minikube/files directory, and Minikube copied it to the /files directory inside the VM.

In some cases, files might end up being copied to the VM’s root (/), instead of to /files. If this has happened to you, please enter the VM (minikube ssh), and move the files to /files, by executing the commands that follow (**only if the /files directory does not exist or is empty**).1

minikube ssh

sudo mkdir /files

sudo mv /prometheus-conf.yml  /files/

exit

📝 Run the above commands one by one.

The time has come to take a look at the content of the file.

minikube ssh sudo chmod +rw \

    /files/prometheus-conf.yml

minikube ssh cat \

    /files/prometheus-conf.yml

We changed the permissions of the file and displayed its content.

The **output** is as follows.

global:

  scrape\_interval:     15s

scrape\_configs:

  - job\_name: prometheus

    metrics\_path: /prometheus/metrics

    static\_configs:

      - targets:

        - localhost:9090

This configuration is almost identical to what Prometheus uses by default. The only difference is in the metrics\_path, which is now pointing to /prometheus/metrics.

In the next lesson, we will work with the updated configuration of the Prometheus application.