**About this Module**

This lesson discusses the intended audience and the necessary prerequisites of the module.

**We'll cover the following**

* [Who is this module for?](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JE2P7wL0Rkv#Who-is-this-module-for?)
* [Module Structure](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JE2P7wL0Rkv#Module-Structure)

**Who is this module for?**

This module is for anyone who has knowledge of how a virtual machine works and wants to get started with Kubernetes.

**Module Structure**

This module contains *fifteen* chapters, namely:

1. **How Did We Get Here?:**

This chapter tells us the need for Kubernetes and the basic information required to be able to understand Kubernetes fully.

1. **Running Kubernetes Cluster Locally:**

This chapter teaches us how to install Kubernetes on our systems and introduces us to the commands we’ll be using with Kubernetes.

1. **Pods:**

This chapter introduces us to pods and how to define and run them. It also teaches us other concepts related to pods such as components, stages, and how to run multiple containers in a single pod.

1. **ReplicaSets:**

This chapter introduces us to ReplicaSets and how to create and operate them.

1. **Services:**

This chapter introduces us to services. It also teaches us how to create services and the concepts that will be required to run the services.

1. **Deployments:**

This chapter teaches us about deploying Kubernetes releases. It also teaches us about updating deployments and zero-downtime deployments.

1. **Ingress:**

This chapter introduces us to Ingress and how it works. It also teaches us the concepts of Ingress and how to create Ingress resources.

1. **Volumes:**

This chapter introduces us to volumes and teaches us how to use volumes for different purposes such as accessing the host’s resources and injecting configuration files.

1. **ConfigMaps:**

This chapter introduces us to ConfigMaps and teaches us how to use them to inject configuration files. It also teaches us how to define ConfigMaps as YAML.

1. **Secrets:**

This chapter gets us started with secrets and goes through built-in secrets. It also teaches us how to create generic secrets.

1. **Namespaces:**

This chapter introduces us to cluster divisions and teaches us how to deploy a release. It also teaches us how to create, deploy, and communicate between namespaces.

1. **Securing Kubernetes Clusters:**

This chapter goes through the security of a Kubernetes cluster. It teaches us access to the Kubernetes API, authorization, and roles.

1. **Managing Resources:**

This chapter teaches us how to manage the resources of a Kubernetes cluster, including container memory and CPU resources.

1. **Creating a Production-Ready Kubernetes Cluster:**

This chapter introduces us to production-ready clusters and takes us through the process of making these clusters.

1. **Persisiting State:**

This chapter introduces us to stateful persistence and deploying stateful applications. It also teaches us how to deal with stateful applications.

# A Short History of Infrastructure Management

Learn a short history of infrastructure management and how we get here.

**We'll cover the following**

* [The beginning](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7n0lWyLO15w#The-beginning)
* [⚙️ Configuration management and relevant tools](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7n0lWyLO15w#%E2%9A%99%EF%B8%8F-Configuration-management-and-relevant-tools)
  + [👍 Pros](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7n0lWyLO15w#%F0%9F%91%8D-Pros)
  + [👎 Cons](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7n0lWyLO15w#%F0%9F%91%8E-Cons)
* [Virtual machines](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7n0lWyLO15w#Virtual-machines)
  + [Mutability vs. immutability](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7n0lWyLO15w#Mutability-vs.-immutability)
* [☁️ The Cloud hosting](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7n0lWyLO15w#%E2%98%81%EF%B8%8F-The-Cloud-hosting)
* [Modern infrastructure](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7n0lWyLO15w#Modern-infrastructure)

## 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 and relevant tools

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.

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

### 👍 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

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

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

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).

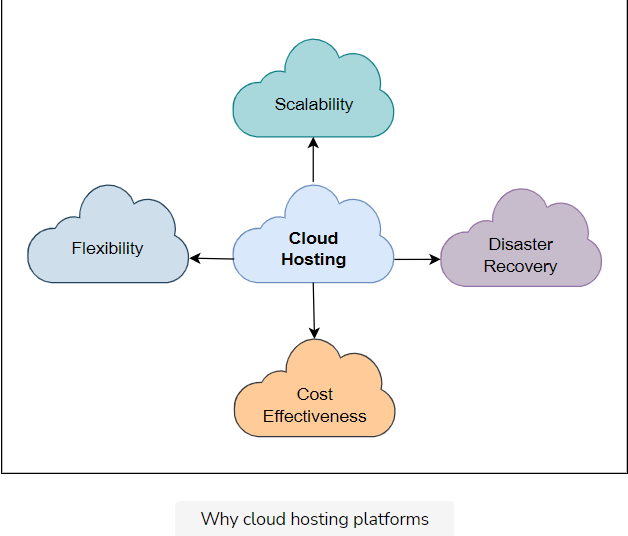
When ideas behind immutable infrastructure started getting traction, people began combining them with the concepts behind configuration management. However, tools available at that time were not fit for the job. They (Chef, Puppet, Ansible, and the like) were designed with the idea that servers are brought 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

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.

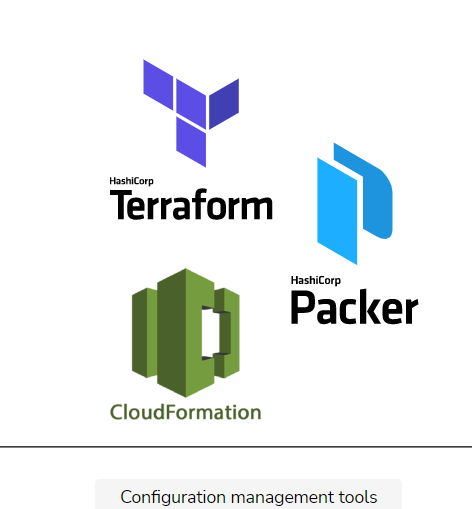


They showed us what can be done. While some companies embraced the cloud, others went into defensive positions. “We can build an internal cloud”, “AWS is too expensive”, “I would, but I can’t because of legislation”, and “our market is different”, are only a few ill-conceived excuses often given by people 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

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.

# A Short History of Deployment Processes

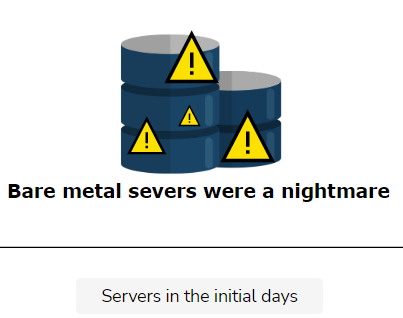
Learn about a short history of deployment processes.

**We'll cover the following**

* [The beginning](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3Yxm6l0NP0n#The-beginning)
* [Configuration management tools](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3Yxm6l0NP0n#Configuration-management-tools)
* [The need of the hour](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3Yxm6l0NP0n#The-need-of-the-hour)
* [Docker and containers](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3Yxm6l0NP0n#Docker-and-containers)
* [Why container schedulers?](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3Yxm6l0NP0n#Why-container-schedulers?)

## 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

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 services 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.

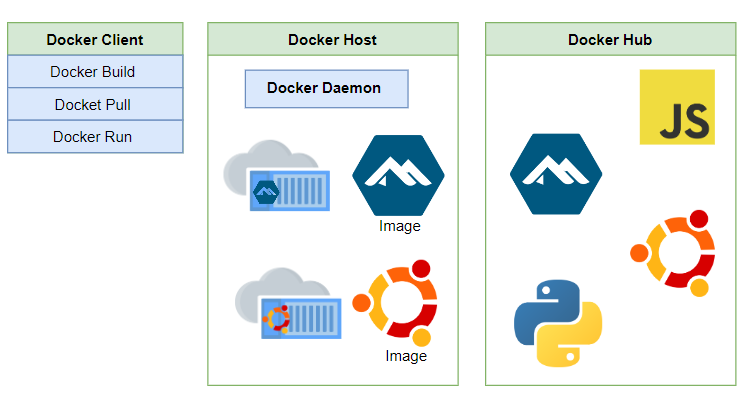


## The need of the hour

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

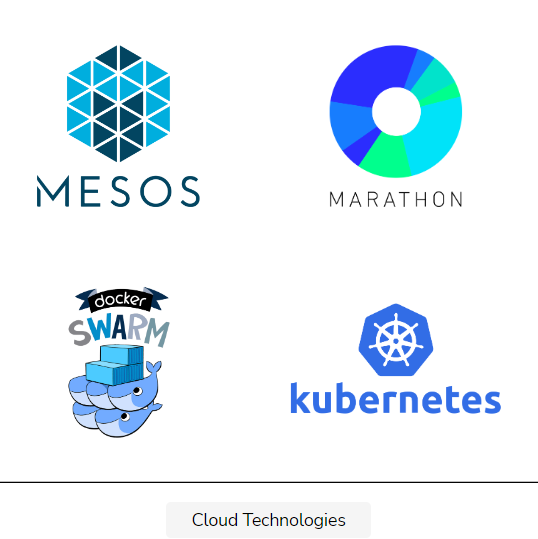
Fortunately, Linux got namespaces, cgroups, and other things that are together known as containers. They were lightweight, fast, and cheap. They provided process isolation and quite a few other benefits. Unfortunately, they were not easy to use. Even though they’ve been around for a while, only a handful 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?

It’s not enough to run containers. We need to be able to scale them, make them fault-tolerant, 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

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

**We'll cover the following**

* [An analogy](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qZ9Og3MJpZD#An-analogy)
* [The relatable part](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qZ9Og3MJpZD#The-relatable-part)
* [Why use schedulers?](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qZ9Og3MJpZD#Why-use-schedulers?)
  + [The container schedulers](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qZ9Og3MJpZD#The-container-schedulers)
    - [Why to combine containers and schedulers?](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qZ9Og3MJpZD#Why-to-combine-containers-and-schedulers?)

## An analogy

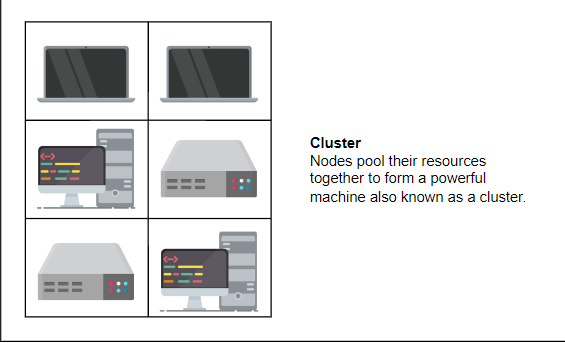
Picture some young teenagers. After school, they’d go to a courtyard and play soccer. That was an exciting sight. A random number of kids running around the yard without any orchestration. There was no offense and no defense. They’d just run after a ball.

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.



## The relatable part

Those soccer 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?

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.

* In an imperative way, you tell Kubernetes what it needs to do and how he needs to do it.
* In a declarative way, you’ll tell Kubernetes what you need and it will do it for you.

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.

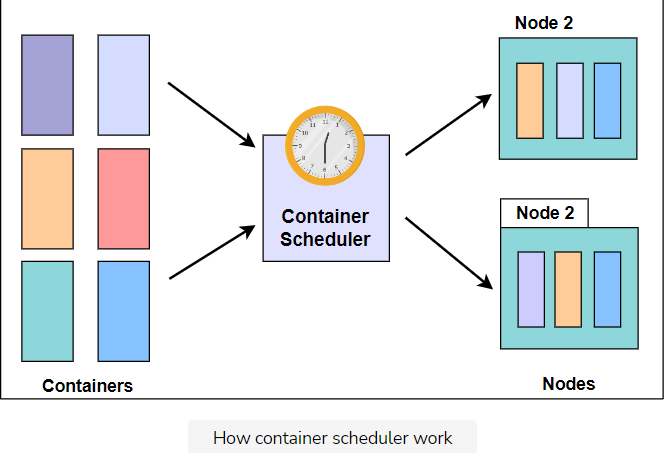
Think of schedulers as operators who are continually monitoring the system and fixing discrepancies between the desired and the actual state. The difference is that schedulers are infinitely faster and more accurate. They do not get tired, they do not need to go to the bathroom, and they do not require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

### The container schedulers

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?

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.

# What is Kubernetes?

Get introduced to the world of Kubernetes.

**We'll cover the following**

* [Don’t run containers directly](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/RLgx4DD37qz#Don%E2%80%99t-run-containers-directly)
* [Kubernetes](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/RLgx4DD37qz#Kubernetes)
  + [Why Kubernetes?](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/RLgx4DD37qz#Why-Kubernetes?)
* [Concluding remarks](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/RLgx4DD37qz#Concluding-remarks)

## Don’t 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.

## 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?

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

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!**

# Getting Started with Pods

Learn what are Pods, and how to create a Kubernetes cluster.

**We'll cover the following**

* [Understanding pods](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/q27l1GOK9L2#Understanding-pods)
* [Creating a cluster](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/q27l1GOK9L2#Creating-a-cluster)

## 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.

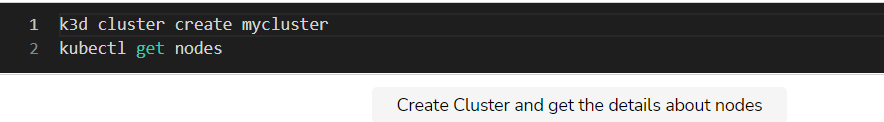
Since we cannot create Pods without a Kubernetes cluster, our first order of business is to create one.

## Creating a cluster

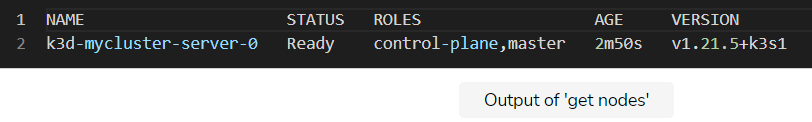
In this course, we will use k3d clusters on our platform. k3d is an open source wrapper to run lightweight k3s clusters on docker containers.

For practicing this course on the Educative Platform all necessary toolkits and installations are already taken care of. However, if you want to experiment with clusters on your local machines, please refer to the **appendix** section of this course.

We’ll create a Kubernetes cluster using k3d, the first command will create a k3d cluster named mycluster.



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



Run the above commands in the Terminal below. Make sure to connect the Terminal by pressing **Click to Connect** before executing any commands.

# A Quick and Dirty way to Run Pods

Learn to create Pods using through the imperative method.

**We'll cover the following**

* [Creating a pod with Mongo](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3Y8yo1JGj89#Creating-a-pod-with-Mongo)
* [Why this is a dirty way?](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3Y8yo1JGj89#Why-this-is-a-dirty-way?)
  + [Troubleshooting tips for minikube](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3Y8yo1JGj89#Troubleshooting-tips-for-minikube)

## Creating a pod with Mongo

Just as we can execute docker run to create containers, kubectl allows us to create Pods with a **single command**. For example, if we’d like to create a Pod with a Mongo database, the command is as follows.

**kubectl run db --image mongo**

You’ll notice that the output says that “pod/db was created”. We created our first Pod. We can confirm that by listing all the Pods in the cluster.

**kubectl get pods**

The **output** should be something like this.

**NAME READY STATUS RESTARTS AGE**

**db 0/1 ContainerCreating 0 1m**

In the **output**, we can see:

* The name of the Pod
* Its readiness
* The status
* The number of times it restarted
* For how long it has existed (its age)

If you were fast enough, or your network is slow, none of the pods might be ready. We expect to have **one** Pod, but there’s **zero** running at the moment.

Since the mongo image is relatively big, it might take a while until it is pulled from *Docker Hub*. After a while, we can retrieve the Pods one more time to confirm that the Pod with the Mongo database is running. The **output** this time is as follows.

**NAME READY STATUS RESTARTS AGE**

**db 1/1 Running 0 6m**

We can see that, this time, the Pod is ready and we can start using the Mongo database.

We can confirm that a container based on the mongo image is indeed running inside the cluster by listing all the containers based on the mongo image.

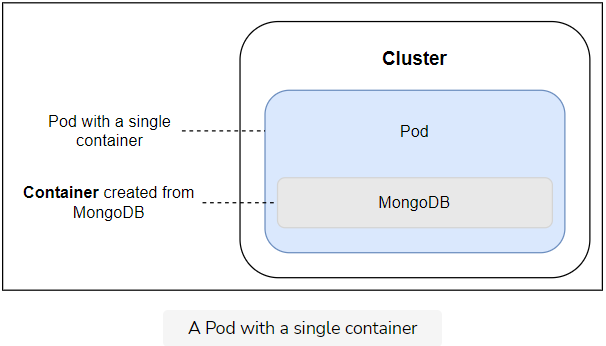
docker exec -it k3d-mycluster-server-0 ctr container ls | grep mongo

The **output** of the above command is as follows (IDs are removed for brevity):

**CONTAINER ID IMAGE**

**... docker.io/library/mongo:latest**

As you can see, the container defined in the Pod is **running**.



That was not the best way to run Pods so we’ll delete it.

**kubectl delete pod db**

The **output** is as follows.

**pod "db" deleted**

For ease of use, a list of all the commands used in this lesson is also given below:

**kubectl run db --image mongo**

**kubectl get pods**

**docker exec -it k3d-mycluster-server-0 ctr container ls | grep mongo**

**kubectl delete pod db**

Run the above commands in the Terminal below. Make sure to connect the Terminal by pressing Click to Connect before executing any commands.

## Why this is a dirty way?

The above approach used to run Pods is not the best one. We used the imperative way to tell Kubernetes what to do. Even though there are cases when that might be useful, most of the time we want to leverage the declarative approach.

We want to have a way to define what we need in a file and pass that information to Kubernetes. That way, we can have a documented and repeatable process, that can (and should) be version controlled as well.

Moreover, the kubectl run was reasonably simple. In real life, we need to declare much more than the name of the deployment and the image. Commands like kubectl can quickly become too long and, in many cases, very complicated. Instead, we’ll write specifications in YAML format.

### Troubleshooting tips for minikube

If you are following along locally with the help of minikube, you will need to execute the following command to get the containers running inside docker:

**eval $(minikube docker-env)**

**docker container ls -f ancestor=mongo**

The **output** of the latter command is as follows (IDs are removed for brevity)

**CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES**

**... mongo "docker-entrypoint.s…" 5 minutes ago Up 5 minutes k8s\_db\_db-...**

# Defining Pods through Declarative Syntax

Learn to create pods using declarative syntax.

**We'll cover the following**

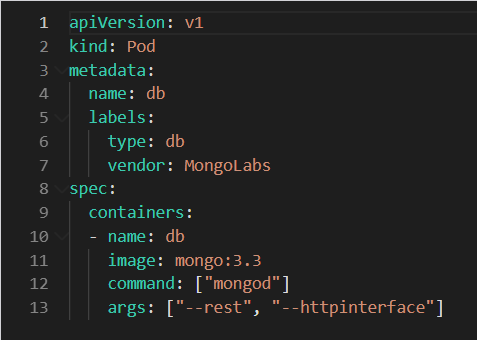
* [Pods as a wrapper to containers](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7XmRDGNMNl1#Pods-as-a-wrapper-to-containers)
* [Looking into a pod’s definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7XmRDGNMNl1#Looking-into-a-pod%E2%80%99s-definition)

## Pods as a wrapper to containers

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

Let us look at a simple Pod db.yml definition.



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/v1.24/#pod-v1-core) documentation.

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



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 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.

**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.

**kubectl get pods -o wide**

The **output** is as follows.

**NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES**

**db 1/1 Running 0 92s 10.42.0.9 k3d-mycluster-server-0 <none> <none**

As you can see, we got four additional columns; the IP, NODE, NOMINATED NODE and READINESS GATE.

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

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.

kubectl get pods -o yaml

Just as with the json output, we won’t go into details of everything we got from yaml. 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.

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.

...

Events:

Type Reason Age From Message

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

Normal Scheduled 4m28s default-scheduler Successfully assigned default/db to k3d-mycluster-server-0

Normal Pulling 4m28s kubelet Pulling image "mongo:3.3"

Normal Pulled 3m15s kubelet Successfully pulled image "mongo:3.3" in 1m13.409798522s

Normal Created 3m11s kubelet Created container db

Normal Started 3m11s kubelet Started container db

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.

For your ease, all the commands used in this lesson are also given below.

**kubectl create -f db.yml**

**kubectl get pods**

**kubectl get pods -o wide**

**kubectl get pods -o json**

**kubectl get pods -o yaml**

**kubectl describe pod db**

# Components and Stages involved in a Pod's Scheduling

Learn about the different stages involved in a pod's creation.

**We'll cover the following**

* [Step-by-step pod’s scheduling](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7ArAKyxDGxA#Step-by-step-pod%E2%80%99s-scheduling)
  + [1. API server](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7ArAKyxDGxA#1.-API-server)
  + [2. Scheduler](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7ArAKyxDGxA#2.-Scheduler)
  + [3. Kubelet](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7ArAKyxDGxA#3.-Kubelet)
* [Sequential breakdown of events](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7ArAKyxDGxA#Sequential-breakdown-of-events)

## Step-by-step pod’s 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

The API server is the central component of a Kubernetes cluster and it runs on the master node. Since we are using k3d, 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

The scheduler is also running on the master node. Its job is to watch for unassigned pods and assign them to a node that 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

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.

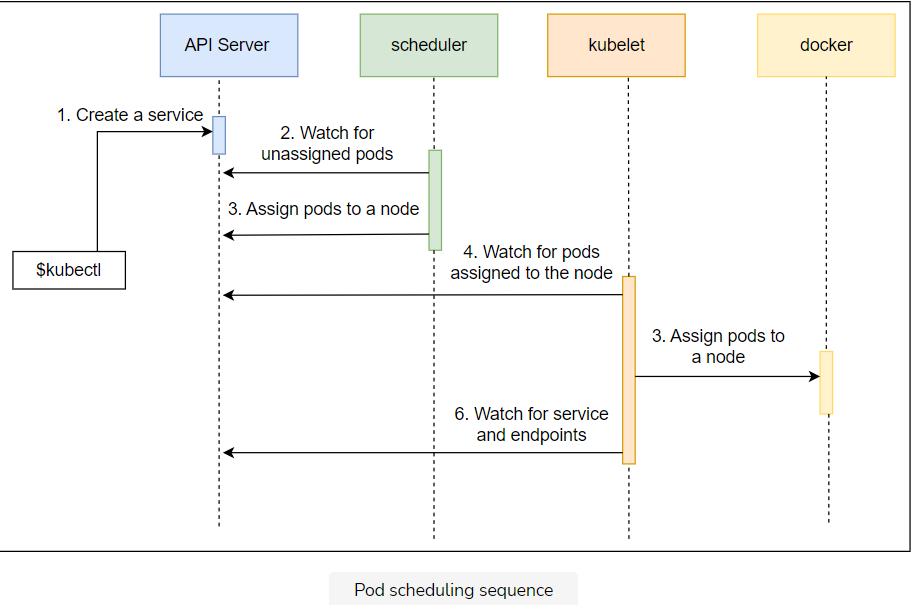
## Sequential breakdown of events

The sequence of events that transpired with the kubectl create -f 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 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.



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

# Playing around with the Running Pod

Learn to explore more about the running pod in the cluster.

**We'll cover the following**

* [Describing resources](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7no445Y6Oy8#Describing-resources)
* [Executing a new process](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7no445Y6Oy8#Executing-a-new-process)
* [Getting logs](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7no445Y6Oy8#Getting-logs)
* [Exploring the failure](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7no445Y6Oy8#Exploring-the-failure)
  + [Killing the container](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7no445Y6Oy8#Killing-the-container)
  + [Deleting the pod](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7no445Y6Oy8#Deleting-the-pod)

## Describing resources

In many cases, it is more useful to describe resources by referencing the file that defines them. That way there is no confusion nor need to remember the names of resources.

Instead of using kubectl describe pod db we could have executed the command that follows:

kubectl describe -f db.yml

The output should be the same as the previous lesson since, in both cases, kubectl sent a request to Kubernetes API requesting information about the Pod named “db”.

## Executing a new process

Just as with Docker, we can execute a new process inside a running container inside a Pod.

kubectl exec db -- ps aux

The **output** will be similar as follows.

USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND

root 1 0.5 2.9 967452 59692 ? Ssl 21:47 0:03 mongod --rest --httpinterface

root 31 0.0 0.0 17504 1980 ? Rs 21:58 0:00 ps aux

We told Kubernetes that we’d like to execute a process inside the first container of the Pod db. Since our Pod defines only one container, this container and the first container are one and the same. The --container (or -c) argument can be set to specify which container should be used. That is particularly useful when running multiple containers in a Pod.

Apart from using Pods as the reference, kubectl exec is almost the same as the docker container exec command. The significant difference is that kubectl allows us to execute a process in a container running in any node inside a cluster, while docker container exec is limited to containers running on a specific node.

Instead of executing a new short-lived process inside a running container, we can enter into it. For example, we can make the execution interactive with -i (stdin) and -t (terminal) arguments and run shell inside a container.

kubectl exec -it db – sh

We’re inside the sh process inside the container. Since the container hosts a Mongo database, we can, for example, execute db.stats() to confirm that the database is indeed running.

echo 'db.stats()' | mongo localhost:27017/test

We used mongo client to execute db.stats() for the database test running on localhost:27017. Since we’re not trying to learn Mongo, the only purpose of this exercise was to prove that the database is up-and-running. Let’s get out of the container.

Exit

## Getting logs

Logs should be shipped from containers to a central location. However, since we did not yet explore that subject, it would be useful to be able to see logs of a container in a Pod.

The command that outputs logs of the only container in the db Pod is as follows:

kubectl logs db

The **output** is too big and not that important in its entirety. One of the last line is as follows.

...

2022-07-18T11:58:45.841+0000 I NETWORK [thread1] waiting for connections on port 27017

...

With the -f (or --follow) we can follow the logs in real-time. Just as with the exec sub-command, if a Pod defines multiple containers, we can specify which one to use with the -c argument.

## Exploring the failure

What happens when a container inside a Pod dies?

### Killing the container

Let’s simulate a failure and observe what happens.

kubectl exec db -- pkill mongod

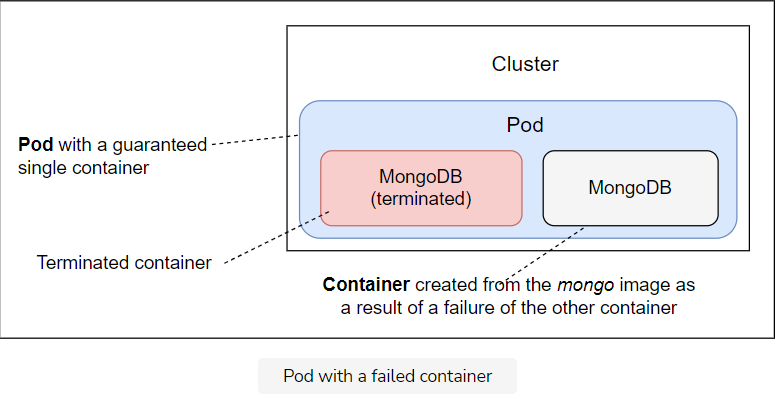
kubectl get pods

We killed the main process of the container and listed all the Pods. The **output** is as follows.

NAME READY STATUS RESTARTS AGE

db 1/1 Running 1 13m

The container is running (1/1). Kubernetes guarantees that the containers inside a Pod are (almost) always running. Please note that the RESTARTS field now has the value of 1. Every time a container fails, Kubernetes will restart it.



### Deleting the pod

Finally, we can delete a Pod if we don’t need it anymore.

kubectl delete -f db.yml

kubectl get pods

We removed the Pods defined in db.yml and retrieved the list of all the Pods in the cluster. The **output** of the latter command is as follows.

NAME READY STATUS RESTARTS AGE

db 0/1 Terminating 1 3h

The number of ready containers dropped to 0, and the status of the “db” Pod is “terminating”.

When we sent the instruction to delete a Pod, Kubernetes tried to terminate it gracefully.

* The first thing it did was to send the TERM(terminate) signal to all the main processes inside the containers that form the Pod.
* From there on, Kubernetes gives each container a period of thirty seconds so that the processes in those containers can shut down gracefully.
* Once the grace period expires, the KILL signal is sent to terminate all the main processes forcefully and, with them, all the containers. The default grace period can be changed through the gracePeriodSeconds value in YAML definition or --grace-period argument of the kubectl delete command.

If we repeat the get pods command thirty seconds after we issued the delete instruction, the Pod should be removed from the system.

kubectl get pods

This time, the **output** is different.

No resources found.

The only Pod we had in the system is no more!

For your ease, all the instructions used in this lesson are also given below:

kubectl describe -f db.yml

kubectl exec db -- ps aux

kubectl exec -it db -- sh

echo 'db.stats()' | mongo localhost:27017/test

exit

kubectl logs db

kubectl exec db -- pkill mongod

kubectl get pods

kubectl delete -f db.yml

kubectl get pods

# Running Multiple Containers in a Single Pod

Learn to run a pod having multiple containers.

**We'll cover the following**

* [Anatomy of a pod](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7AJjGmvZ72A#Anatomy-of-a-pod)
* [Formatting the Output](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7AJjGmvZ72A#Formatting-the-Output)
* [Executing commands inside the pod](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7AJjGmvZ72A#Executing-commands-inside-the-pod)

## Anatomy of a pod

* **Pods** are designed to run multiple cooperative processes that should act as a cohesive unit. Those processes are wrapped in containers.
* All the containers that form a Pod are running on the same machine. A Pod cannot be split across multiple nodes.
* All the processes (containers) inside a Pod share the same set of resources, and they can communicate with each other through localhost. One of those shared resources is storage.
* A volume (think of it as a directory with shareable data) defined in a Pod can be accessed by all the containers thus allowing them all to share the same data.

We’ll explore storage and volumes in more depth later on. For now, let’s take a look at the go-demo-2.yml specification.

apiVersion: v1

kind: Pod

metadata:

name: go-demo-2

labels:

type: stack

spec:

containers:

- name: db

image: mongo:3.3

- name: api-1

image: vfarcic/go-demo-2

env:

- name: DB

value: localhost

- name: api-2

image: vfarcic/go-demo-2

env:

- name: DB

value: localhost

The db.yml file defines a Pod with two containers named db and api. The service inside the vfarcic/go-demo-2 image uses environment variable DB to know where the database is. The value is localhost since all the containers in the same Pod are reachable through it. Let’s create the Pod.

kubectl create -f go-demo-2.yml

kubectl get -f go-demo-2.yml

We created a new Pod defined in the go-demo-2.yml file and retrieved its information from Kubernetes. The **output** of the latter command is as follows.

NAME READY STATUS RESTARTS AGE

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

We can see from the READY column that, this time, the Pod has two containers (2/2).

This might be an excellent opportunity to introduce formatting to retrieve specific information.

## Formatting the Output

Let’s say that we want to retrieve the names of the containers in a Pod. The first thing we’d have to do is get familiar with Kubernetes API. We can do that by going to [Pod v1 core](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.24/#pod-v1-core) documentation. While reading the documentation will become mandatory sooner or later, we’ll use a simpler route and inspect the output from Kubernetes.

kubectl get -f go-demo-2.yml -o json

The output is too big to be presented here, so we’ll focus on the task at hand. We need to retrieve the names of the containers in the Pod. Therefore, the part of the **output** we’re looking for is as follows.

{

...

"spec": {

"containers": [

{

...

"name": "db",

...

},

{

...

"name": "api",

...

}

],

...

},

...

}

The get command that would filter the output and retrieve only the names of the containers is as follows.

kubectl get -f go-demo-2.yml \

-o jsonpath="{.spec.containers[\*].name}"

The **output** is as follows.

db api

We used jsonpath as the output format and specified that we want to retrieve names of all the containers from the spec. The ability to filter and format information might not look that important right now but, once we move into more complex scenarios, it will prove to be invaluable. That will become especially evident when we try to automate the processes and requests sent to Kubernetes API.

## Executing commands inside the pod[#](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7AJjGmvZ72A#Executing-commands-inside-the-pod)

How would we execute a command inside the Pod? Unlike the previous examples that did a similar task, this time we have two containers in the Pod, so we need to be more specific.

kubectl exec -it -c db go-demo-2 -- ps aux

The **output** should display the processes inside the db container. Namely, the mongod process.

How about logs from a container? As you might have guessed, we cannot execute something like kubectl logs go-demo-2 since the Pod hosts multiple containers. Instead, we need to be specific and name the container from which we want to see the logs.

kubectl logs go-demo-2 -c db

How about scaling? How would we, for example, scale the service so that there are two containers of the API and one container for the database?

One option could be to define two containers in the Pod. Let’s take a look at a Pod definition that might accomplish what we need.

apiVersion: v1

kind: Pod

metadata:

name: go-demo-2

labels:

type: stack

spec:

containers:

- name: db

image: mongo:3.3

- name: api-1

image: vfarcic/go-demo-2

env:

- name: DB

value: localhost

- name: api-2

image: vfarcic/go-demo-2

env:

- name: DB

value: localhost

We defined two containers for the API and named them api-1 and api-2. The only thing left is to create the Pod. Before creating pod using the updated definition, let us first delete the pod created earlier.

kubectl delete -f go-demo-2.yml

Delete pod

For ease of use, all of the commands used in this lesson are provided below:

kubectl create -f go-demo-2.yml

kubectl get -f go-demo-2.yml

kubectl get -f go-demo-2.yml -o json

kubectl get -f go-demo-2.yml -o jsonpath="{.spec.containers[\*].name}"

kubectl exec -it -c db go-demo-2 -- ps aux

kubectl logs go-demo-2 -c db

kubectl delete -f go-demo-2.yml

# Single vs. Multi-Container Pods

Learn about the single-container and multi-container pods.

**We'll cover the following**

* [Why single-container pods?](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JPrn3LRLJm2#Why-single-container-pods?)
* [Multi-container pods](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JPrn3LRLJm2#Multi-container-pods)

## Why single-container pods?

We should not think of Pods as resources that should do anything beyond the 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)**
* Continuous Delivery **(CD)**
* Continuous Deployment processes **(CDP)**

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

# Monitoring Health

Learn why it is important to monitor the health of services and how Kubernetes Probes can help to achieve this.

**We'll cover the following**

* [Why to monitor health?](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qZvMo3MGXPk#Why-to-monitor-health?)
* [Kubernetes Probes](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qZvMo3MGXPk#Kubernetes-Probes)
  + [Liveness Probe](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qZvMo3MGXPk#Liveness-Probe)
  + [Readiness Probe](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qZvMo3MGXPk#Readiness-Probe)
* [Understanding the updated pod definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qZvMo3MGXPk#Understanding-the-updated-pod-definition)
  + [Liveness probe in action](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qZvMo3MGXPk#Liveness-probe-in-action)
* [Pods are (almost) useless (by themselves)](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qZvMo3MGXPk#Pods-are-(almost)-useless-(by-themselves))
* [Destroying Everything](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qZvMo3MGXPk#Destroying-Everything)
* [Troubleshooting tips for minikube](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qZvMo3MGXPk#Troubleshooting-tips-for-minikube)

## Why to monitor health?

The go-demo-2 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

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

### 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

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

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

apiVersion: v1

kind: Pod

metadata:

name: go-demo-2

labels:

type: stack

spec:

containers:

- name: db

image: mongo:3.3

- name: api

image: vfarcic/go-demo-2

env:

- name: DB

value: localhost

livenessProbe:

httpGet:

path: /this/path/does/not/exist

port: 8080

initialDelaySeconds: 5

timeoutSeconds: 2 # Defaults to 1

periodSeconds: 5 # Defaults to 10

failureThreshold: 1 # Defaults to 3

* **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

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

kubectl -f 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.

kubectl describe -f go-demo-2-health.yml

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

Events:

Type Reason Age From Message

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

Normal Scheduled 70s default-scheduler Successfully assigned default/go-demo-2 to k3d-mycluster-server-0

Normal Pulling 69s kubelet Pulling image "mongo:3.3"

Normal Pulled 41s kubelet Successfully pulled image "mongo:3.3" in 28.738663826s

Normal Created 39s kubelet Created container db

Normal Started 39s kubelet Started container db

Normal Pulled 36s kubelet Successfully pulled image "vfarcic/go-demo-2" in 2.975792483s

Normal Pulled 29s kubelet Successfully pulled image "vfarcic/go-demo-2" in 387.506265ms

Normal Started 19s (x3 over 36s) kubelet Started container api

Normal Pulled 19s kubelet Successfully pulled image "vfarcic/go-demo-2" in 375.450308ms

Warning Unhealthy 10s (x3 over 30s) kubelet Liveness probe failed: HTTP probe failed with statuscode: 404

Normal Killing 10s (x3 over 30s) kubelet Container api failed liveness probe, will be restarted

Normal Pulling 10s (x4 over 39s) kubelet Pulling image "vfarcic/go-demo-2"

Normal Pulled 9s kubelet Successfully pulled image "vfarcic/go-demo-2" in 375.064683ms

Normal Created 9s (x4 over 36s) kubelet Created container api

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/v1.24/#pod-v1-core) if you’d like to learn all the available options.

## 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.

For your ease, all the instructions have been given below:

kubectl create -f go-demo-2-health.yml

kubectl describe -f go-demo-2-health.yml

You can practice the commands in the following code playground by pressing the Run button and waiting for the cluster to set up.

## Destroying Everything

We’ll remove the cluster and start the next chapter fresh. Delete the cluster using using the command below:

k3d cluster delete mycluster –all

## Troubleshooting tips for minikube[#](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qZvMo3MGXPk" \l "Troubleshooting-tips-for-minikube)

If you are working with minikube, use the following instruction to delete the cluster.

minikube delete

# What's Next?

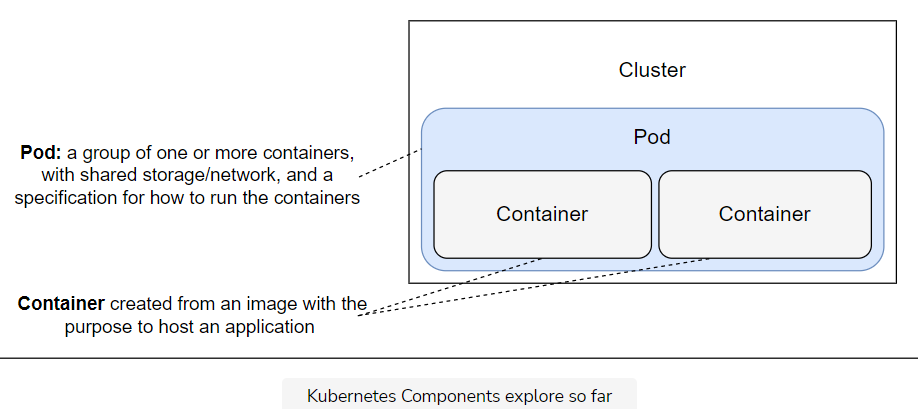
Reiterate what we have learned so far about pods.

**We'll cover the following**

* [Destroying the cluster](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7DpBK65DMjy#Destroying-the-cluster)

## Destroying the cluster

Please take some time to get more familiar with Pods. They are the most basic and, arguably, the essential building block in Kubernetes. Since, by now, you have a solid understanding of what Pods are, a good next step might be to go through [PodSpec v1 core](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.24/" \l "podspec-v1-core" \t "_blank) documentation.



The above illustration shows and defines the Kubernetes components we have explored so far.

# Getting Started with ReplicaSets

Learn what are ReplicaSets and what is their significance in Kubernetes.

**We'll cover the following**

* [Understanding ReplicaSets](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/YQLk2KPxx80#Understanding-ReplicaSets)

## 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 service are (almost) always running.

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

# Creating ReplicaSets

Learn how to create and retrieve a replicaSet

**We'll cover the following**

* [Looking into the definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3Yko9Zpz65Q#Looking-into-the-definition)
* [Creating the replicaSet](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3Yko9Zpz65Q#Creating-the-replicaSet)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3Yko9Zpz65Q#Try-it-yourself)

## Looking into the definition

Let’s take a look at a ReplicaSet based on the Pod we created in the previous chapter go-demo-2.yml.

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

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

kubectl create -f 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.

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 go-demo-2.yml file.

kubectl get -f 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 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

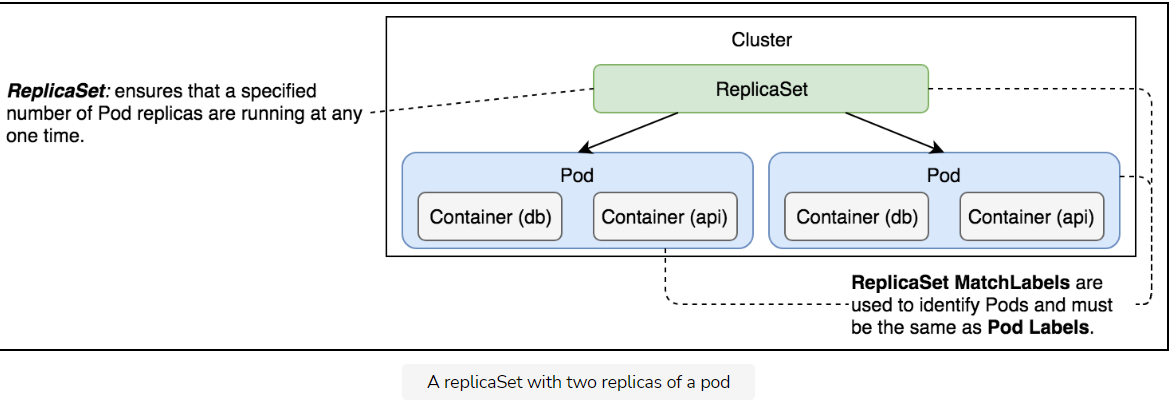
To be on the safe side, we used the --show-labels argument so that we can verify that the Pods in the cluster match those created by the ReplicaSet.

The **output** is as follows.

NAME READY STATUS RESTARTS AGE LABELS

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



## Try it yourself

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

For your ease, all the instructions in the lessons have been given below:

kubectl create -f go-demo-2.yml

kubectl get rs

kubectl get -f go-demo-2.yml

kubectl describe -f go-demo-2.yml

kubectl get pods --show-labels

# Sequential Breakdown of the Process

Learn the sequence of events involved in the process of creating a replicaSet.

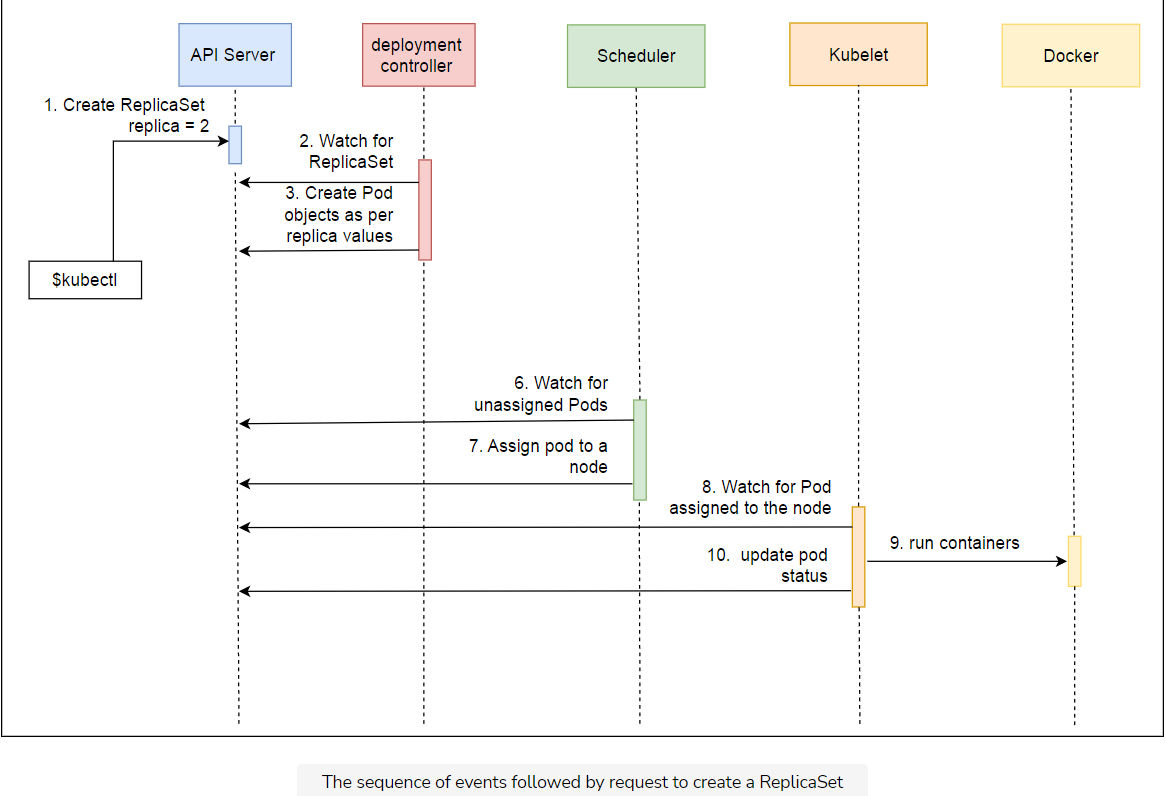
**We'll cover the following**

* [Step by step process](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gxBg0qgl9wl#Step-by-step-process)

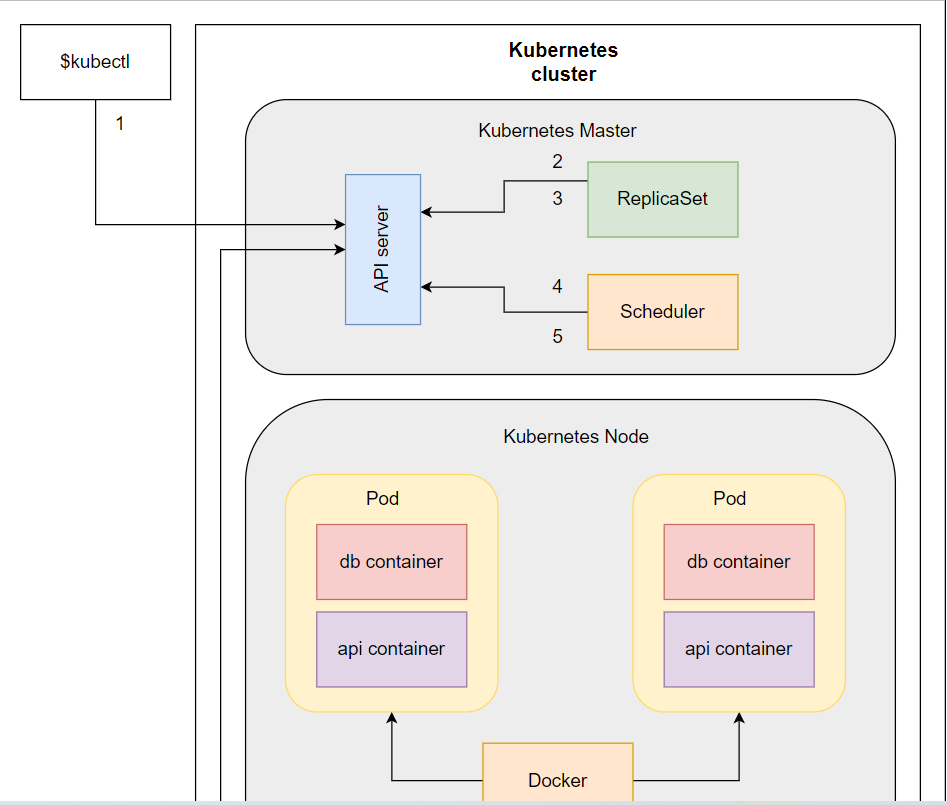
## Step by step process

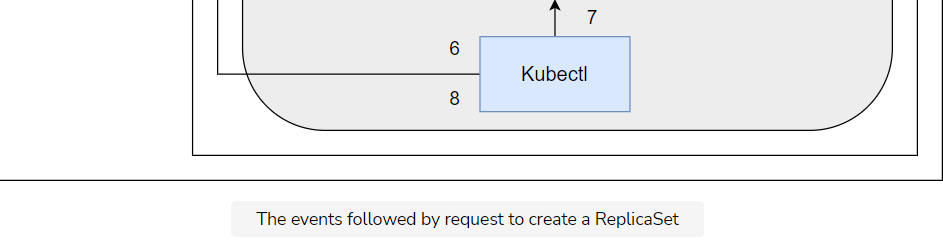
The sequence of events that transpired with the kubectl create -f go-demo-2.yml command is as follows.

1. Kubernetes client (kubectl) sent a request to the API server requesting the creation of a ReplicaSet defined in the 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 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.



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.





Typically, we’d have a multi-node cluster, and the Pods would be distributed across it. For now, while we’re using k3d, 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.

# Operating ReplicaSets

Explore the operating procedure of ReplicaSets to witness its self-healing property in action.

**We'll cover the following**

* [Deleting ReplicaSets](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7nP3mwP8RVj#Deleting-ReplicaSets)
* [Re-using the same pods](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7nP3mwP8RVj#Re-using-the-same-pods)
* [Updating the definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7nP3mwP8RVj#Updating-the-definition)
* [Self-healing in action](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7nP3mwP8RVj#Self-healing-in-action)
  + [Destroying a pod](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7nP3mwP8RVj#Destroying-a-pod)
  + [Removing a label](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7nP3mwP8RVj#Removing-a-label)
  + [Re-adding the label](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7nP3mwP8RVj#Re-adding-the-label)
* [Destroying Everything](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7nP3mwP8RVj#Destroying-Everything)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7nP3mwP8RVj#Try-it-yourself)

## 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 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 go-demo-2.yml --cascade=orphan

We used the --cascade=orphan 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=orphan 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

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 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

Since we saved the configuration, we can apply an updated definition of the ReplicaSet. For example, we can use 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 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

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

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

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

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.

## Destroying Everything

We will destroy the cluster using the command below before moving onto the next chapter.

k3d cluster delete mycluster –all

## Try it yourself

All of the commands used in this section are given below.

kubectl delete -f go-demo-2.yml --cascade=orphan

kubectl get rs

kubectl get pods

kubectl create -f go-demo-2.yml --save-config

kubectl get pods

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

kubectl get pods

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

| tail -1)

kubectl delete $POD\_NAME

kubectl get pods

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

| tail -1)

kubectl label $POD\_NAME service-

kubectl describe $POD\_NAME

kubectl get pods --show-labels

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

kubectl get pods --show-labels

k3d cluster delete mycluster –all

You can practice the commands in the following code playground by pressing the Run button and waiting for the cluster to set up.

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

# What's Next?

Recap what was discussed in this chapter.

**We'll cover the following**

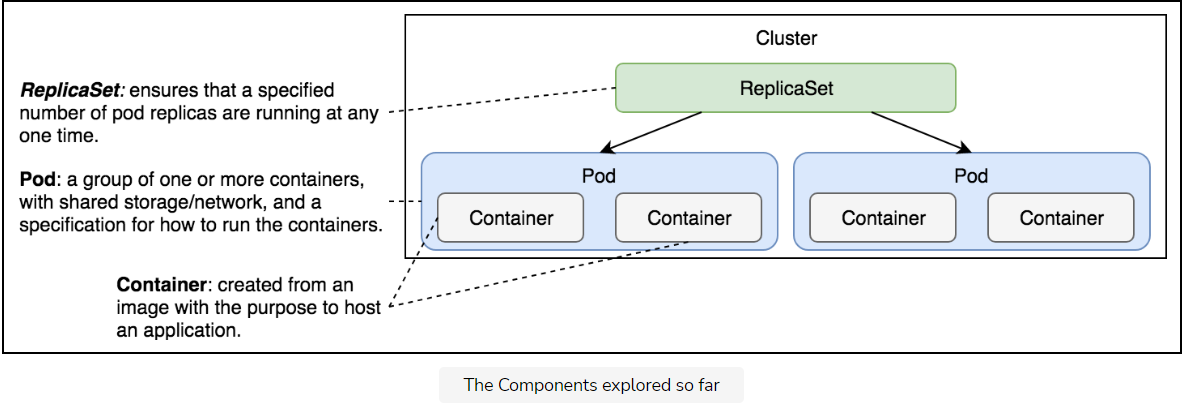
* [Summary](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qVXvDNKMEDp#Summary)

## Summary

The good news is that ReplicaSets are relatively straightforward. They provide a guarantee that the specified number of replicas of a Pod will be running in the system as long as there are available resources. That’s the primary and, arguably, the only purpose.

The bad news is that ReplicaSets are rarely used independently. You will almost never create a ReplicaSet directly just as you’re not going to create Pods. Instead, we tend to create ReplicaSets through Deployments. In other words, we use ReplicaSets to create and control Pods, and Deployments to create ReplicaSets (and a few other things). We’ll get to Deployment soon.

ℹ️ If you’d like to know more about ReplicaSets, please explore [ReplicaSet v1 apps](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.24/" \l "replicaset-v1-apps" \t "_blank) API documentation.



The above illustration shows and defines the Kubernetes components we have explored so far.

# Getting Started with Communication

Find out why establishing communication between pods is necessary.

**We'll cover the following**

* [The Problem](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N8BA5Mk0VKK#The-Problem)
* [The Solution](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N8BA5Mk0VKK#The-Solution)

## 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

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

# Creating Services by Exposing Ports

Learn how to create Kubernetes Services by exposing ports.

**We'll cover the following**

* [Creating ReplicaSets](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qVMx5kYYoyR#Creating-ReplicaSets)
* [Exposing a resource](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qVMx5kYYoyR#Exposing-a-resource)
* [Other types of services](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qVMx5kYYoyR#Other-types-of-services)
  + [ClusterIP](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qVMx5kYYoyR#ClusterIP)
  + [LoadBalancer](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qVMx5kYYoyR#LoadBalancer)
  + [ExternalName](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qVMx5kYYoyR#ExternalName)
* [Try it youself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qVMx5kYYoyR#Try-it-youself)

## 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 go-demo-2-rs. 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 go-demo-2-rs.yml

kubectl get -f 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

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

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

### 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

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

### 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.

## Try it youself

All the commands used in this lesson are given below.

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

kubectl get -f go-demo-2-rs.yml

kubectl expose rs go-demo-2 \

--name=go-demo-2-svc \

--target-port=28017 \

--type=NodePort

**root@8059c028005f2c54:/usercode# kubectl get svc**

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

**kubernetes ClusterIP 10.43.0.1 <none> 443/TCP 8m55s**

**go-demo-2-svc NodePort 10.43.41.168 <none> 28017:30265/TCP 5m41s**

# Sequential Breakdown of the Process

Learn the sequential processes kicked off by a Service creation.

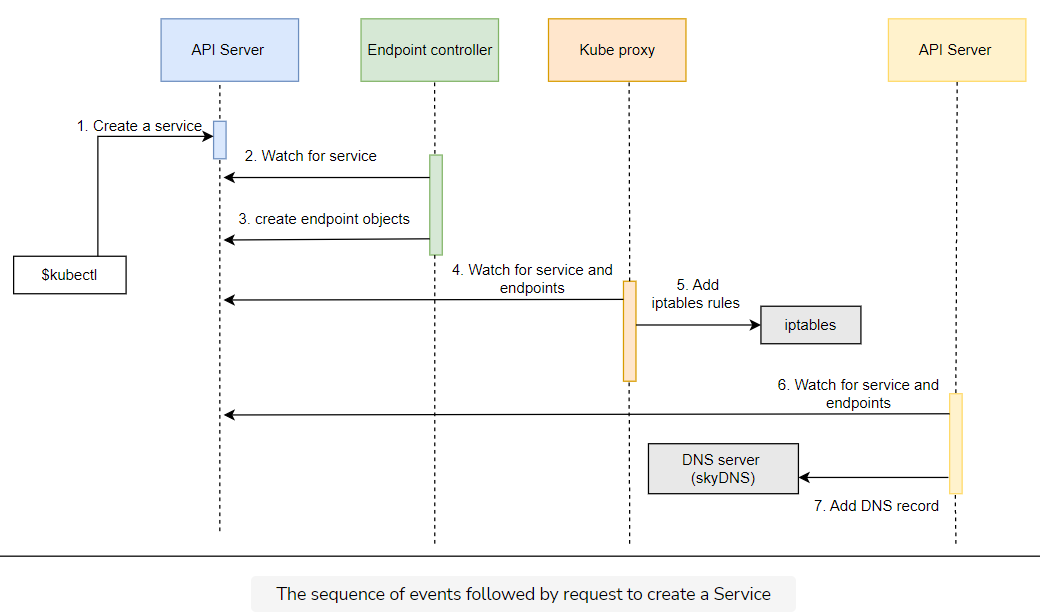
**We'll cover the following**

* [The sequence](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JY7jKEXPVBl#The-sequence)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JY7jKEXPVBl#Try-it-yourself)

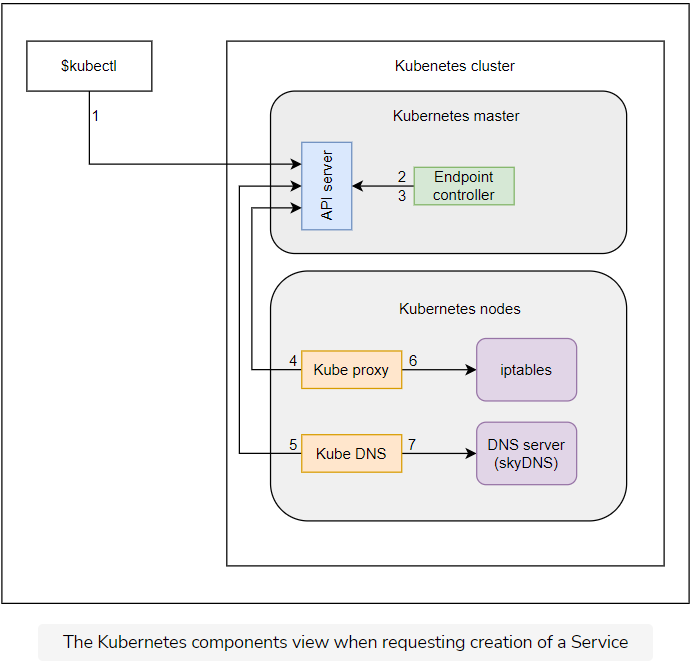
## The sequence

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 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.



Let’s take a look at our new Service.

kubectl describe svc/go-demo-2-svc

Name: go-demo-2-svc

Namespace: default

Labels: <none>

Annotations: <none>

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

Type: NodePort

IP Family Policy: SingleStack

IP Families: IPv4

IP: 10.43.126.144

IPs: 10.43.126.144

Port: <unset> 28017/TCP

TargetPort: 28017/TCP

NodePort: <unset> 30754/TCP

Endpoints: 10.42.0.7:28017,10.42.0.8:28017

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 6:** 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.

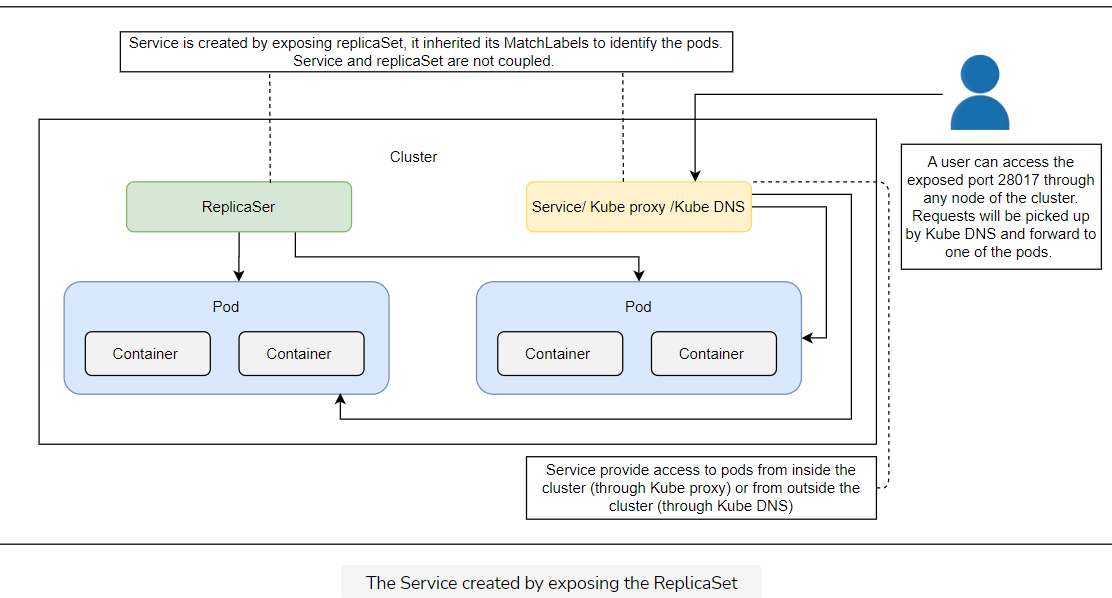
kubectl port-forward service/go-demo-2-svc 3000:28017 --address 0.0.0.0

#Now click on the link beside Run button

# Exit the port-forward process by clicking Ctrl + C for Windows

# Or Control + C for Mac before running the next command

After running the above command click on the link beside the run button to see the UI of mongodb opened in the browser. Make sure that the pods are ready before running the this command.



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

## Try it yourself

A list of commands used in this lesson are also provided below.

kubectl describe svc/go-demo-2-svc

kubectl port-forward service/go-demo-2-svc 3000:28017 --address 0.0.0.0

#now click the link beside the run button

# Exit the port-forward process by clicking Ctrl + C for Windows

# Or Control + C for Mac before running the next command

kubectl delete svc go-demo-2-svc

# Creating Services through Declarative Syntax

Learn to create Services through declarative syntax.

**We'll cover the following**

* [Looking into the syntax](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/q2J1r0o0xL2#Looking-into-the-syntax)
* [Creating the service](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/q2J1r0o0xL2#Creating-the-service)
* [Request forwarding](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/q2J1r0o0xL2#Request-forwarding)
* [Now we can split](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/q2J1r0o0xL2#Now-we-can-split)
* [Destroying everything](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/q2J1r0o0xL2#Destroying-everything)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/q2J1r0o0xL2#Try-it-yourself)

## Looking into the syntax

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

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

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

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

kubectl get -f 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.

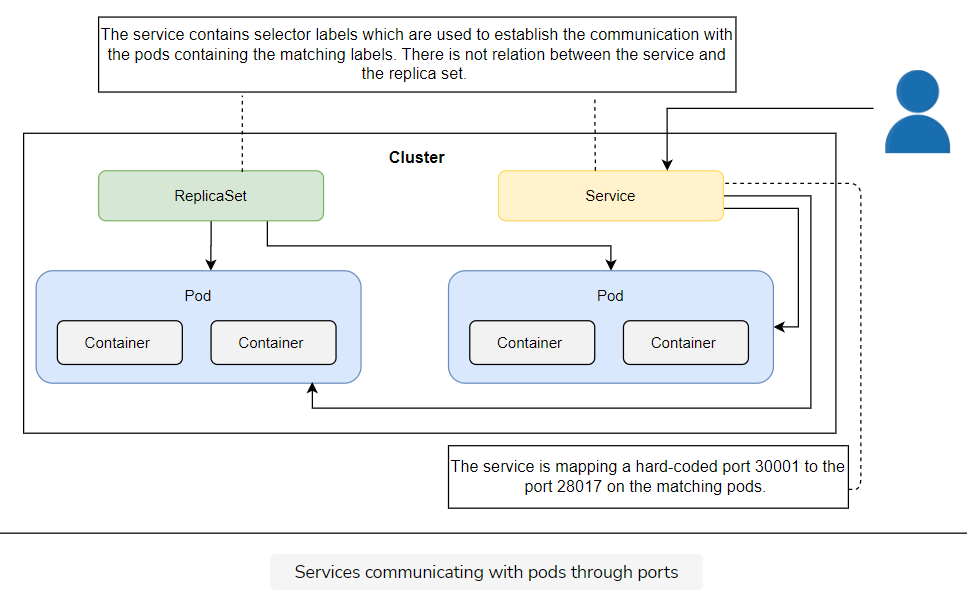
kubectl port-forward service/go-demo-2 3000:28017 --address 0.0.0.0

#Now click on the link beside Run button

# Exit the port-forward process by clicking Ctrl + C for Windows

# Or Control + C for Mac before running the next command

After running the above command click on the link beside the run button to see the UI of mongodb opened in the browser. Make sure that the pods are ready before running the above command.



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:

annotations:

endpoints.kubernetes.io/last-change-trigger-time: "2022-07-22T07:29:03Z"

creationTimestamp: "2022-07-22T07:29:03Z"

name: go-demo-2

namespace: default

resourceVersion: "848"

uid: 24246fb5-cc9c-49a9-bf69-2d55218e4326

subsets:

- addresses:

- ip: 10.42.0.10

nodeName: k3d-mycluster-server-0

targetRef:

kind: Pod

name: go-demo-2-9gmps

namespace: default

resourceVersion: "808"

uid: 88065f5e-cc5f-42b9-bd08-1d6ffa5301ef

- ip: 10.42.0.9

nodeName: k3d-mycluster-server-0

targetRef:

kind: Pod

name: go-demo-2-5l86n

namespace: default

resourceVersion: "811"

uid: 556a3219-c0f7-4f87-82ed-0feb3c8b88a8

ports:

- port: 28017

protocol: TCP

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

## 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

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

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

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

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

## Try it yourself

A list of all the commands used in this lesson is given below

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

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

kubectl port-forward service/go-demo-2 3000:28017 --address 0.0.0.0

#Now click on the link beside Run button

# Exit the port-forward process by clicking Ctrl + C for Windows

# Or Control + C for Mac before running the next command

kubectl get ep go-demo-2 -o yaml

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

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

# Splitting the Pod and Establishing communication through Services

Learn how to split the Pods, create a separate DB pod and a Service to communicate with it.

**We'll cover the following**

* [Looking into the definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N7LOzQ3gzxN#Looking-into-the-definition)
* [Creating the ReplicaSet](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N7LOzQ3gzxN#Creating-the-ReplicaSet)
* [Creating the service](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N7LOzQ3gzxN#Creating-the-service)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N7LOzQ3gzxN#Try-it-yourself)

## Looking into the definition

Let’s take a look at a ReplicaSet definition go-demo-2-db-rs.yml for a Pod with only the database.

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

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

kubectl create -f go-demo-2-db-rs.yml

One object was created, three are left to go.

## Creating the service

The next one is the Service go-demo-2-db-svc.yml for the Pod we just created through the ReplicaSet.

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 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.

## Try it yourself

The list of all the commands used in this lesson is given below.

kubectl create -f go-demo-2-db-rs.yml

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

# Creating the Split API Pods

Learn to create API Pods using ReplicaSet and establish communication by creating Service.

**We'll cover the following**

* [Looking into the definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xVk0y1PEg0r#Looking-into-the-definition)
* [The readinessProbe](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xVk0y1PEg0r#The-readinessProbe)
* [Creating the ReplicaSet](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xVk0y1PEg0r#Creating-the-ReplicaSet)
* [Creating the Service](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xVk0y1PEg0r#Creating-the-Service)
* [Accessing the API](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xVk0y1PEg0r#Accessing-the-API)
* [Destroying services](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xVk0y1PEg0r#Destroying-services)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xVk0y1PEg0r#Try-it-yourself)
  + [Troubleshooting tips for minikube](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xVk0y1PEg0r#Troubleshooting-tips-for-minikube)

## Looking into the definition

Let’s see the definition of backend API go-demo-2-api-rs.yml.

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: go-demo-2-api

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

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

Now let’s create the ReplicaSet go-demo-2-api-rs.yml.

kubectl create -f go-demo-2-api-rs.yml

## Creating the Service

Only one object is missing, that is Service go-demo-2-api-svc.yml, the definition is given below.

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 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

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.

nohup kubectl port-forward service/go-demo-2-api --address 0.0.0.0 3000:8080 > /dev/null 2>&1 &

# Please wait a few seconds before executing the following command

curl -i <http://localhost:3000/demo/hello>

The **output** of the last command is as follows. You can also open the link beside run button to see the page.

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

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

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

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

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

kubectl delete -f 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.

## Try it yourself[#](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xVk0y1PEg0r#Try-it-yourself)

A list of all the commands used in this lesson is given below.

kubectl create -f go-demo-2-api-rs.yml

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

kubectl get all

nohup kubectl port-forward service/go-demo-2-api --address 0.0.0.0 3000:8080 > /dev/null 2>&1 &

#Please wait a few seconds before running the following command

curl -i "0.0.0.0:3000/demo/hello"

### Troubleshooting tips for minikube**[#](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xVk0y1PEg0r" \l "Troubleshooting-tips-for-minikube)**

You won’t always need to bind the ports using the port-forward command to interact with the services. If you are using minikube, you can use the following commands to interact with the service:

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

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

IP=$(minikube ip)

curl -i "http://$IP:$PORT/demo/hello"

# Discovering Services

Learn the process of discovering Services.

**We'll cover the following**

* [Discovering services using DNS and environment variables](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/YMqADL0BoJ9#Discovering-services-using-DNS-and-environment-variables)
* [Sequential breakdown of the process](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/YMqADL0BoJ9#Sequential-breakdown-of-the-process)
* [Destroying Everything](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/YMqADL0BoJ9#Destroying-Everything)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/YMqADL0BoJ9#Try-it-yourself)

## Discovering services using DNS and environment variables

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 i.e. go-demo-2-db.

Let's get environment variables for go-demo-2-db

POD\_NAME=$(kubectl get pod \

--no-headers \

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

-l type=db,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.

PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

HOSTNAME=go-demo-2-db-dgbqg

GOSU\_VERSION=1.7

MONGO\_MAJOR=3.3

MONGO\_VERSION=3.3.15

MONGO\_PACKAGE=mongodb-org-unstable

KUBERNETES\_PORT=tcp://10.43.0.1:443

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

GO\_DEMO\_2\_DB\_PORT\_27017\_TCP\_ADDR=10.43.223.206

GO\_DEMO\_2\_API\_PORT\_8080\_TCP=tcp://10.43.54.204:8080

GO\_DEMO\_2\_API\_PORT\_8080\_TCP\_PORT=8080

GO\_DEMO\_2\_API\_PORT\_8080\_TCP\_ADDR=10.43.54.204

KUBERNETES\_SERVICE\_PORT\_HTTPS=443

KUBERNETES\_PORT\_443\_TCP\_PORT=443

KUBERNETES\_PORT\_443\_TCP\_ADDR=10.43.0.1

GO\_DEMO\_2\_API\_SERVICE\_HOST=10.43.54.204

GO\_DEMO\_2\_API\_PORT=tcp://10.43.54.204:8080

KUBERNETES\_SERVICE\_HOST=10.43.0.1

GO\_DEMO\_2\_DB\_SERVICE\_HOST=10.43.223.206

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

GO\_DEMO\_2\_DB\_PORT=tcp://10.43.223.206:27017

GO\_DEMO\_2\_DB\_PORT\_27017\_TCP=tcp://10.43.223.206:27017

GO\_DEMO\_2\_API\_SERVICE\_PORT=8080

KUBERNETES\_SERVICE\_PORT=443

KUBERNETES\_PORT\_443\_TCP=tcp://10.43.0.1:443

KUBERNETES\_PORT\_443\_TCP\_PROTO=tcp

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

GO\_DEMO\_2\_API\_PORT\_8080\_TCP\_PROTO=tcp

HOME=/root

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 Family Policy: SingleStack

IP Families: IPv4

IP: 10.43.164.13

IPs: 10.43.164.13

Port: <unset> 27017/TCP

TargetPort: 27017/TCP

Endpoints: 10.42.0.9:27017

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.

...

env:

- name: DB

value: go-demo-2-db

...

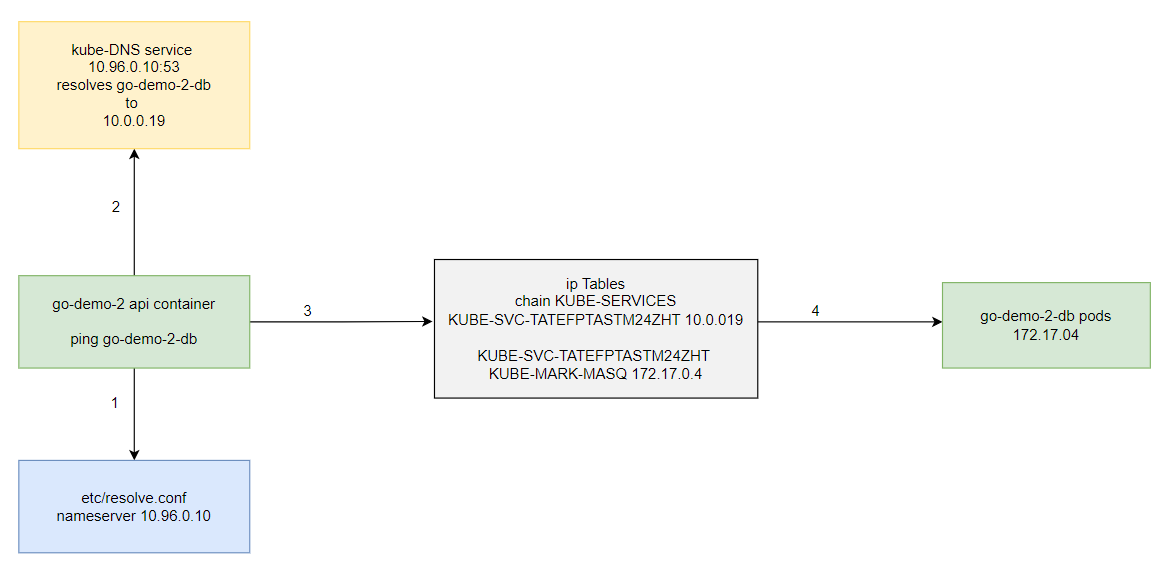
We declared an environment variable with the name of the Service (go-demo-2-db). That variable is used by the code as a connection string to the database.

Kubernetes converts Service names into DNSes and adds them to the DNS server.

## 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.



## Destroying Everything

We have exhausted this topic and the time has come to destroy everything we did so far. Use the following command to delete the cluster to start afresh for the next chapter.

k3d cluster delete mucluster –all

## Try it yourself

A list of commands used in this lesson is given below.

# Comparison with Docker Swarm

Compare Kubernetes Pods, ReplicaSets, and Services with Docker Swarm equivalents.

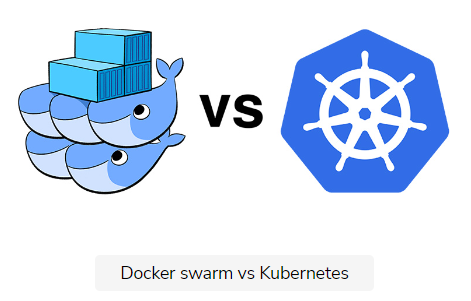
**We'll cover the following**

* [Comparing Pods, ReplicaSets, and services](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/YQx22PXwVy9#Comparing-Pods,-ReplicaSets,-and-services)
  + [Looking into Kubernetes definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/YQx22PXwVy9#Looking-into-Kubernetes-definition)
  + [Looking into Docker Swarm sefinition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/YQx22PXwVy9#Looking-into-Docker-Swarm-sefinition)
  + [Pods](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/YQx22PXwVy9#Pods)
  + [ReplicaSets](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/YQx22PXwVy9#ReplicaSets)
  + [Services](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/YQx22PXwVy9#Services)
  + [Conclusion](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/YQx22PXwVy9#Conclusion)

## Comparing Pods, ReplicaSets, and services

Starting from this chapter, we’ll compare each Kubernetes feature with Docker Swarm equivalents. That way, Swarm users can have a smoother transition into Kubernetes or, depending on their goals, choose to stick with Swarm.

Please bear in mind that the comparisons will be made only for a specific set of features. You will not (yet) be able to conclude whether Kubernetes is better or worse than Docker Swarm. You’ll need to grasp both products in their entirety to make an educated decision. The comparisons like those that follow are useful only as a base for more detailed examinations of the two products.



For now, we’ll limit the comparison scope to Pods, ReplicaSets, and Services on the one hand, and Docker Service stacks, on the other.

### Looking into Kubernetes definition

Let’s start with Kubernetes file go-demo-2.yml (the same one we used before).

The definition 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

spec:

containers:

- name: db

image: mongo:3.3

ports:

- containerPort: 28017

---

apiVersion: v1

kind: Service

metadata:

name: go-demo-2-db

spec:

ports:

- port: 27017

selector:

type: db

service: go-demo-2

---

apiVersion: apps/v1beta2

kind: ReplicaSet

metadata:

name: go-demo-2-api

spec:

replicas: 3

selector:

matchLabels:

type: api

service: go-demo-2

template:

metadata:

labels:

type: api

service: go-demo-2

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

---

apiVersion: v1

kind: Service

metadata:

name: go-demo-2-api

spec:

type: NodePort

ports:

- port: 8080

selector:

type: api

service: go-demo-2

### Looking into Docker Swarm sefinition

Now, let’s take a look at the Docker stack defined in go-demo-2-swarm.yml.

The specification is as follows.

version: "3"

services:

api:

image: vfarcic/go-demo-2

environment:

- DB=db

ports:

- 8080

deploy:

replicas: 3

db:

image: mongo

### Pods

Both definitions accomplish the same result. There is no important difference from the functional point of view, except in Pods. Docker does not have the option to create something similar. When Swarm services are created, they are spread across the cluster, and there is no easy way to specify that multiple containers should run on the same node. Whether multi-container Pods are useful or not is something we’ll explore later. For now, we’ll ignore that feature.

### ReplicaSets

If we execute something like docker stack deploy -c go-demo-2-swarm.yml go-demo-2, the result would be equivalent to what we got when we run kubectl create -f go-demo-2.yml. In both cases, we get three replicas of vfarcic/go-demo-2, and one replica of mongo. Respective schedulers are making sure that the desired state (almost) always matches the actual state. Networking communication through internal DNSes is also established with both solutions. Each node in a cluster would expose a randomly defined port that forwards requests to the api. All in all, there are no functional differences between the two solutions.

### Services

When it comes to the way services are defined, there is indeed, a considerable difference. Docker’s stack definition is much more compact and straight-forward. We defined, in twelve lines, what took around eighty lines in the Kubernetes format.

One might argue that Kubernetes YAML file could have been smaller. Maybe it could. Still, it’ll be bigger and more complex no matter how much we simplify it. One might also say that Docker’s stack is missing readinessProbe and livenessProbe. Yes it is, and that is because we decided not to put it there, because the vfarcic/go-demo-2 image already has HEALTHCHECK definition that Docker uses for similar purposes. In most cases, Dockerfile is a better place to define health checks than a stack definition. That does not mean that it cannot be set, or overwritten, in a YAML file. It can, when needed. But, that is not the case in this example.

### Conclusion

All in all, if we limit ourselves only to Kubernetes Pods, ReplicaSets, and Services, and their equivalents in Docker Swarm, the latter wins due to a much simpler and more straightforward way to define specs. From the functional perspective, both are very similar.

Should you conclude that Swarm is a better option than Kubernetes? Not at all. At least, not until we compare other features. Swarm won the battle, but the war has just begun. As we progress, you’ll see that there’s much more to Kubernetes. We only scratched the surface.

**Q.** When NodePort is used for creating a service, ClusterIP will be created automatically. (True)

Q. Which of the following command can be used to expose a resource as a new Kubernetes Service.

kubectl expose

# What's Next?

Recap what is learned so far and what we are going to learn next.

**We'll cover the following**

* [Summary](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JY0qrMj3gJo#Summary)

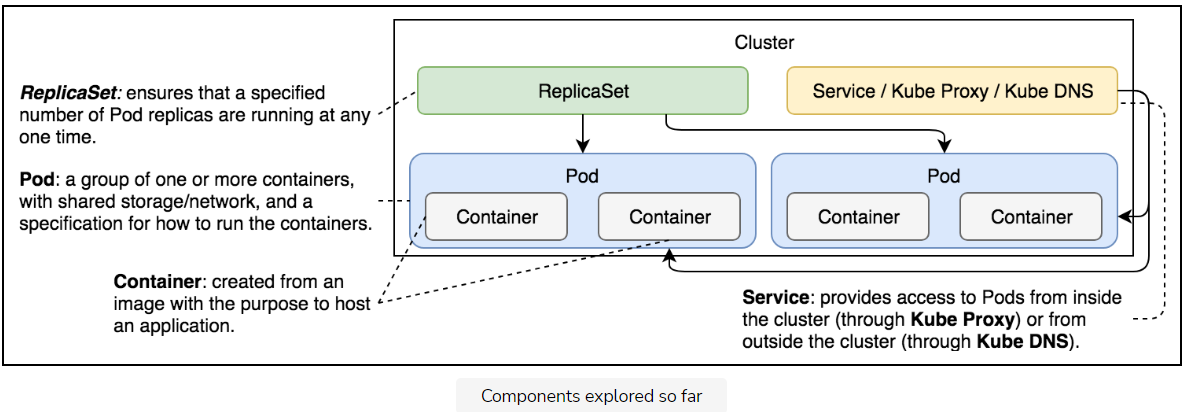
## Summary

That was it. We went through the most important aspects of Services. There are a few other cases we did not yet explore, but the current knowledge should be more than enough to get you going.

Services are indispensable objects without which communication between Pods would be hard and volatile. They provide static addresses through which we can access them not only from other Pods but also from outside the cluster. This ability to have fixed entry points is crucial as it provides stability to otherwise dynamic elements of the cluster. Pods come and go, Services stay.

We are one crucial topic away from having a fully functional, yet still simple, strategy for deployment and management of our applications. We are yet to explore how to deploy and update our services without downtime.

ℹ️ If you’d like to know more about Services, please explore [Service v1 core](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.24/#service-v1-core) API documentation.



The above illustration shows and defines the Kubernetes components we have explored so far.

# Getting Started with Deploying Releases

Learn about releasing features to production with zero downtime.

**We'll cover the following**

* [Deploying releases](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gkV0mBX0n5k#Deploying-releases)
* [Why zero downtime?](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gkV0mBX0n5k#Why-zero-downtime?)
* [Kubernetes deployments](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gkV0mBX0n5k#Kubernetes-deployments)

## 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?

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

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

# Deploying new Releases

Learn Kubernetes deployment definition and create a deployment.

**We'll cover the following**

* [Looking into the definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7nzLG5wJM31#Looking-into-the-definition)
* [Creating the deployment](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7nzLG5wJM31#Creating-the-deployment)
* [Describing the Deployment](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7nzLG5wJM31#Describing-the-Deployment)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7nzLG5wJM31#Try-it-yourself)

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

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

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/module/lesson/a-practical-guide-to-kubernetes/7nzLG5wJM31#Creating-the-deployment)

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

kubectl create \

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

kubectl get -f 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

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

kubectl describe \

-f 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

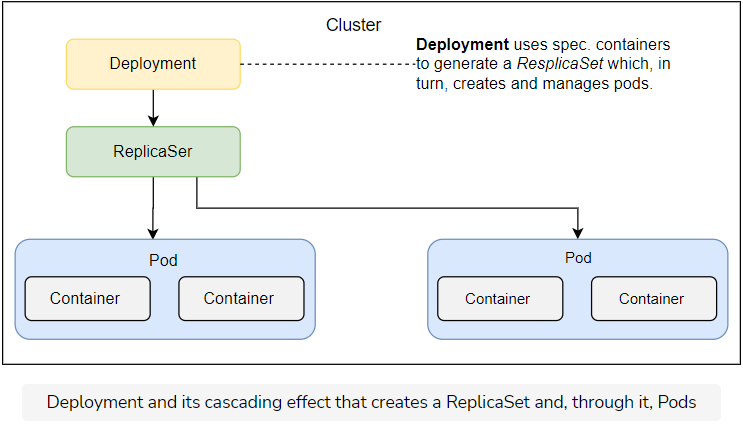
replicaset.apps/go-demo-2-db-694bfb44cb 1 1 1 7m49s

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.



## Try it yourself

The list of all the commands used in this lesson is given below.

kubectl create -f go-demo-2-db.yml

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

kubectl describe -f go-demo-2-db.yml

kubectl get all

# Sequential Breakdown of the Process

Understand the sequential breakdown of the Kubernetes Deployment process.

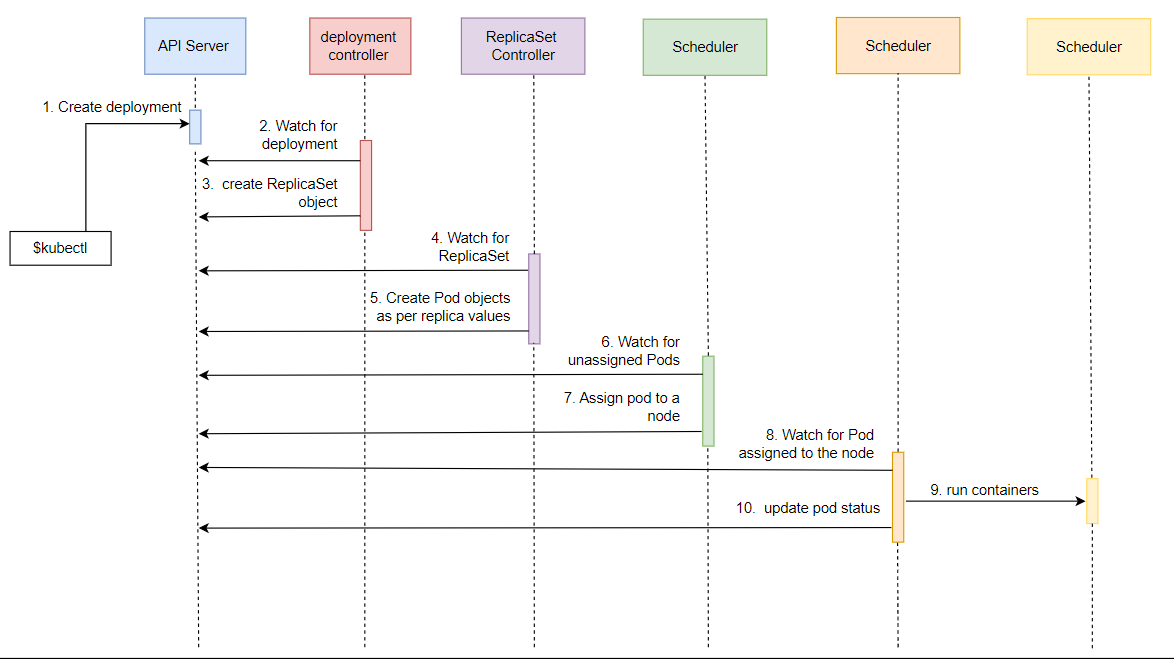
**We'll cover the following**

* [The sequence](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/m7ov2Q0p34O#The-sequence)

## 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 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.



# Updating Deployments

Learn to update the Kubernetes Deployments.

**We'll cover the following**

* [Updating the db image](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3wBjEAWk91p#Updating-the-db-image)
* [Describing the deployment](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3wBjEAWk91p#Describing-the-deployment)
* [Looking into the cluster](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3wBjEAWk91p#Looking-into-the-cluster)
* [Exploring ways to update deployment](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3wBjEAWk91p#Exploring-ways-to-update-deployment)
  + [Updating using commands](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3wBjEAWk91p#Updating-using-commands)
  + [Updating the YAML file](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3wBjEAWk91p#Updating-the-YAML-file)
* [Finishing off by adding a service](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3wBjEAWk91p#Finishing-off-by-adding-a-service)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3wBjEAWk91p#Try-it-yourself)

## Updating the db image

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

kubectl set image \

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

db=mongo:3.4 \

--record

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

## Describing the deployment

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

kubectl describe \

-f go-demo-2-db.yml

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

...

Events:

Type Reason Age From Message

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

Normal ScalingReplicaSet 101s deployment-controller Scaled up replica set go-demo-2-db-76668544d4 to 1

Normal ScalingReplicaSet 11s deployment-controller Scaled up replica set go-demo-2-db-6b48fcbfcf to 1

Normal ScalingReplicaSet 10s deployment-controller Scaled down replica set go-demo-2-db-76668544d4 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

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-6b48fcbfcf-px5wn 1/1 Running 0 5m18s

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

service/kubernetes ClusterIP 10.43.0.1 <none> 443/TCP 7m9s

NAME READY UP-TO-DATE AVAILABLE AGE

deployment.apps/go-demo-2-db 1/1 1 1 6m48s

NAME DESIRED CURRENT READY AGE

replicaset.apps/go-demo-2-db-6b48fcbfcf 1 1 1 5m18s

replicaset.apps/go-demo-2-db-76668544d4 0 0 0 6m48s

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 “6b48fcbfcf”. 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

### 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 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

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

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 go-demo-2-db-svc.yml

## Try it yourself

The list of all the commands used in this lesson is as follows.

kubectl set image -f go-demo-2-db.yml db=mongo:3.4

kubectl describe -f go-demo-2-db.yml

kubectl get all

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

# Defining a Zero-Downtime Deployment

Learn the definition of a zero-downtime deployment.

**We'll cover the following**

* [Looking into the Definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g73KmmoMLjY#Looking-into-the-Definition)
* [Deployment Strategies](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g73KmmoMLjY#Deployment-Strategies)
  + [Recreate strategy](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g73KmmoMLjY#Recreate-strategy)
  + [RollingUpdate strategy](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g73KmmoMLjY#RollingUpdate-strategy)
* [The template](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g73KmmoMLjY#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 go-demo-2 image.

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

## Looking into the Definition

Let’s take a look at the Deployment definition of the API go-demo-api.

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

env:

- name: DB

value: go-demo-2-db

readinessProbe:

httpGet:

path: /demo/hello

port: 8080

periodSeconds: 1

livenessProbe:

httpGet:

path: /demo/hello

port: 8080

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

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

### 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

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

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

Learn how to create a zero-downtime deployment using the file defined in the previous lesson.

**We'll cover the following**

* [Creating deployment](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JEN447QM6pP#Creating-deployment)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JEN447QM6pP#Try-it-yourself)

## Creating deployment[#](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JEN447QM6pP#Creating-deployment)

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

kubectl create -f go-demo-2-api.yml --record

kubectl get -f 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 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 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 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-559d6888df (3/3 replicas created)

Events:

Type Reason Age From Message

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

Normal ScalingReplicaSet 64s deployment-controller Scaled up replica set go-demo-2-api-566c5b97f5 to 3

Normal ScalingReplicaSet 45s deployment-controller Scaled up replica set go-demo-2-api-559d6888df to 1

Normal ScalingReplicaSet 45s deployment-controller Scaled down replica set go-demo-2-api-566c5b97f5 to 2

Normal ScalingReplicaSet 45s deployment-controller Scaled up replica set go-demo-2-api-559d6888df to 2

Normal ScalingReplicaSet 39s deployment-controller Scaled down replica set go-demo-2-api-566c5b97f5 to 0

Normal ScalingReplicaSet 39s deployment-controller Scaled up replica set go-demo-2-api-559d6888df to 3

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-559d6888df”) to 1. Next, it decreased the number of replicas of the old ReplicaSet (“go-demo-2-api-566c5b97f5”) 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 go-demo-2-api.yml

The **output** is as follows.

deployment.apps/go-demo-2-api

REVISION CHANGE-CAUSE

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

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

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?

1

kubectl get rs

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

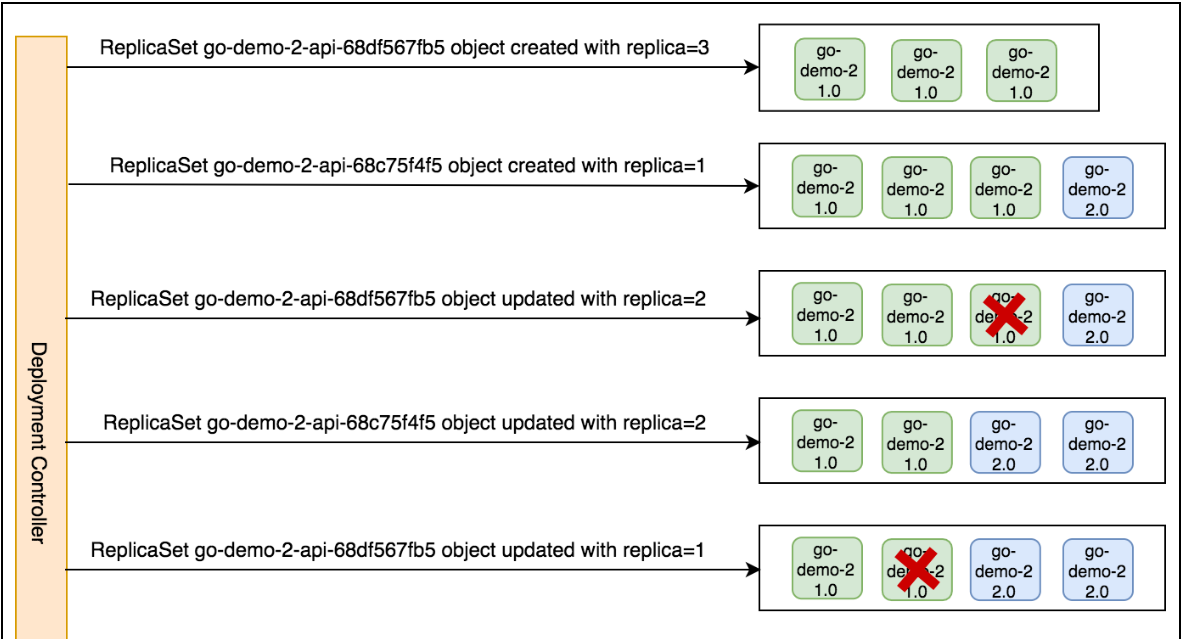
NAME DESIRED CURRENT READY AGE

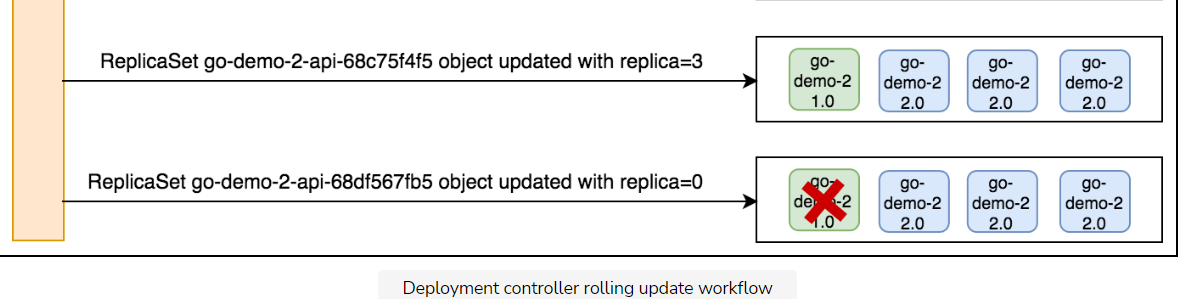
go-demo-2-db-6b48fcbfcf 1 1 1 4m12s

go-demo-2-api-559d6888df 3 3 3 2m5s

...

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.





The above diagram illustrates the workflow of the Deployment controller rolling update.

## Try it yourself

The list of all the commands used in this lesson is given below.

kubectl create -f go-demo-2-api.yml --record

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

kubectl set image -f go-demo-2-api.yml api=vfarcic/go-demo-2:2.0 --record

kubectl rollout status -w -f go-demo-2-api.yml

kubectl describe -f go-demo-2-api.yml

kubectl rollout history -f go-demo-2-api.yml

kubectl get rs

# Rolling Back or Rolling Forward?

Explore different scenarios to decide whether to roll back the Deployment or roll forward.

**We'll cover the following**

* [Understanding the scenarios](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/396O6239oo4#Understanding-the-scenarios)
* [Rolling back with Kubernetes](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/396O6239oo4#Rolling-back-with-Kubernetes)
* [Getting the rollout history](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/396O6239oo4#Getting-the-rollout-history)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/396O6239oo4#Try-it-yourself)

## 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

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. Before going to the command let us first create the cluster and deploy go-demo-api.

kubectl rollout undo -f go-demo-2-api.yml

kubectl describe -f 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-566c5b97f5 (3/3 replicas created)

Events:

Type Reason Age From Message

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

Normal ScalingReplicaSet 4m18s deployment-controller Scaled up replica set go-demo-2-api-559d6888df to 1

Normal ScalingReplicaSet 4m18s deployment-controller Scaled down replica set go-demo-2-api-566c5b97f5 to 2

Normal ScalingReplicaSet 4m18s deployment-controller Scaled up replica set go-demo-2-api-559d6888df to 2

Normal ScalingReplicaSet 2m54s deployment-controller Scaled down replica set go-demo-2-api-566c5b97f5 to 0

Normal ScalingReplicaSet 2m53s deployment-controller Scaled up replica set go-demo-2-api-559d6888df to 3

Normal ScalingReplicaSet 24s deployment-controller Scaled up replica set go-demo-2-api-566c5b97f5 to 1

Normal ScalingReplicaSet 24s deployment-controller Scaled down replica set go-demo-2-api-559d6888df to 2

Normal ScalingReplicaSet 24s deployment-controller Scaled up replica set go-demo-2-api-566c5b97f5 to 2

Normal ScalingReplicaSet 21s deployment-controller (combined from similar events): Scaled down replica set go-demo-2-api-559d6888df to 0

Normal ScalingReplicaSet 21s (x2 over 4m19s) deployment-controller Scaled up replica set go-demo-2-api-566c5b97f5 to 3

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

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 go-demo-2-api.yml

The **output** is as follows.

deployment.apps/go-demo-2-api

REVISION CHANGE-CAUSE

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

3 kubectl create --filename=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 not 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.

## Try it yourself

The code used in this lesson is given below.

kubectl rollout undo -f go-demo-2-api.yml

kubectl describe -f go-demo-2-api.yml

kubectl rollout history -f go-demo-2-api.yml

# Playing around with the Deployment

Explore deploying a few new releases and play around with the Deployment to explore multiple options.

**We'll cover the following**

* [Deploying new releases](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/myzN5WNYvO3#Deploying-new-releases)
* [Checking the rollout history](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/myzN5WNYvO3#Checking-the-rollout-history)
* [Undoing the rollout](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/myzN5WNYvO3#Undoing-the-rollout)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/myzN5WNYvO3#Try-it-yourself)

## 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. As always before going to the details, let us make the necessary deployment e.g. go-demo-2-api.

kubectl set image -f go-demo-2-api.yml api=vfarcic/go-demo-2:3.0 --record

kubectl rollout status -f 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 go-demo-2-api.yml api=vfarcic/go-demo-2:4.0 --record

kubectl rollout status -f 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

Now that we deployed a few releases, we can check the current rollout history.

kubectl rollout history -f go-demo-2-api.yml

The **output** is as follows.

deployment.apps/go-demo-2-api

REVISION CHANGE-CAUSE

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

3 kubectl create --filename=go-demo-2-api.yml --record=true

4 kubectl set image api=vfarcic/go-demo-2:3.0 --filename=go-demo-2-api.yml --record=true

5 kubectl set image api=vfarcic/go-demo-2:4.0 --filename=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

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 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 go-demo-2-api.yml --to-revision=2

kubectl rollout history -f 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.

deployment.apps/go-demo-2-api

REVISION CHANGE-CAUSE

3 kubectl create --filename=go-demo-2-api.yml --record=true

4 kubectl set image api=vfarcic/go-demo-2:3.0 --filename=go-demo-2-api.yml --record=true

5 kubectl set image api=vfarcic/go-demo-2:4.0 --filename=go-demo-2-api.yml --record=true

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

## Try it yourself

A list of all the commands used in this lesson is given below.

kubectl set image -f go-demo-2-api.yml api=vfarcic/go-demo-2:3.0 --record

kubectl rollout status -f go-demo-2-api.yml

kubectl set image -f go-demo-2-api.yml api=vfarcic/go-demo-2:4.0 --record

kubectl rollout status -f go-demo-2-api.yml

kubectl rollout history -f go-demo-2-api.yml

kubectl rollout undo -f go-demo-2-api.yml --to-revision=2

kubectl rollout history -f go-demo-2-api.yml

# Rolling Back Failed Deployments

Learn how to roll back the failed deployments.

**We'll cover the following**

* [Why to roll back?](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/B8mwD4mAEKY#Why-to-roll-back?)
* [Verification](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/B8mwD4mAEKY#Verification)
  + [Undo rollout](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/B8mwD4mAEKY#Undo-rollout)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/B8mwD4mAEKY#Try-it-yourself)

## 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.

Let's update the definition of our Image and deploy it again.

kubectl set image -f 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

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 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

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 go-demo-2-api.yml

kubectl rollout status -f 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 go-demo-2-db.yml

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

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

## Try it yourself

A list of all the commands used in this section is given below.

kubectl set image -f go-demo-2-api.yml api=vfarcic/go-demo-2:does-not-exist --record

kubectl get rs -l type=api

kubectl rollout status -f go-demo-2-api.yml

echo $?

kubectl rollout undo -f go-demo-2-api.yml

kubectl rollout status -f go-demo-2-api.yml

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

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

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

# Merging Everything into the Same YAML Definition

Merge all the definitions explored until now into a single YAML file and then create objects using that file.

**We'll cover the following**

* [Looking into the merged file](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xVM7PKv8q5P#Looking-into-the-merged-file)
* [The differences](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xVM7PKv8q5P#The-differences)
* [Creating objects with the merged file](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xVM7PKv8q5P#Creating-objects-with-the-merged-file)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xVM7PKv8q5P#Try-it-yourself)

## 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.

Let us look at the definition of go-demo-2.yml.

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

---

apiVersion: v1

kind: Service

metadata:

name: go-demo-2-db

spec:

ports:

- port: 27017

selector:

type: db

service: go-demo-2

---

apiVersion: apps/v1beta2

kind: Deployment

metadata:

name: go-demo-2-api

labels:

type: api

service: go-demo-2

language: go

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

---

apiVersion: v1

kind: Service

metadata:

name: go-demo-2-api

spec:

type: NodePort

ports:

- port: 8080

selector:

type: api

service: go-demo-2

## 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

Let’s create the objects defined in dgo-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 go-demo-2.yml --record --save-config

kubectl get -f 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.

## Try it yourself

List of all the commands used in this lesson are give below.

kubectl create -f go-demo-2.yml --record --save-config

kubectl get -f go-demo-2.yml

# Updating Multiple Objects

Learn how to update multiple deployments at a time.

**We'll cover the following**

* [The use of selector labels](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xlZwWMk52rq#The-use-of-selector-labels)
* [Defining the deployment](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xlZwWMk52rq#Defining-the-deployment)
* [Creating the deployment](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xlZwWMk52rq#Creating-the-deployment)
* [Updating the deployments](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xlZwWMk52rq#Updating-the-deployments)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xlZwWMk52rq#Try-it-yourself)

## 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

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 of different-app-db.

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

Let’s create the deployment.

kubectl create \

-f 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

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.apps/go-demo-2-db image updated

deployment.apps/different-app-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 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 different-app-db.yml. You should see that its image was also updated to the newer version.

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl create -f different-app-db.yml

kubectl get deployments --show-labels

kubectl get deployments -l type=db,vendor=MongoLabs

kubectl set image deployments -l type=db,vendor=MongoLabs db=mongo:3.4 --record

kubectl describe -f go-demo-2.yml

# Scaling Deployments

Learn to scale deployments using a YAML file and will discuss automated scaling briefly.

**We'll cover the following**

* [Scaling using YAML files](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/q2ZPV2gg65k#Scaling-using-YAML-files)
* [Looking into the file](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/q2ZPV2gg65k#Looking-into-the-file)
* [Applying the definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/q2ZPV2gg65k#Applying-the-definition)
  + [Verification](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/q2ZPV2gg65k#Verification)
* [Automated scaling](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/q2ZPV2gg65k#Automated-scaling)
* [Scaling the deployment](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/q2ZPV2gg65k#Scaling-the-deployment)
  + [Verification](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/q2ZPV2gg65k#Verification)
* [Destroying Everything](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/q2ZPV2gg65k#Destroying-Everything)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/q2ZPV2gg65k#Try-it-yourself)

## 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

Let’s take a look at go-demo-2-scaled.yml for deployments. We won’t display the contents of the whole file since it is almost identical 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

At the moment, we’re running three replicas. Once we apply the new definition, it should increase to five.

kubectl apply -f 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

Let’s confirm that there are indeed five replicas of the Pods controlled through the Deployment.

kubectl get -f 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

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

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

Let’s confirm that scaling indeed worked as expected.

kubectl get -f 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).

## Destroying Everything

Before we enter the next stage of our knowledge-seeking mission, we’ll destroy the cluster we’re running and give our machines a break.

k3d cluster delete mycluster --all

## Try it yourself

A list of all the commands used in the lesson is given below.

#wait a few seconds before executing the first command

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

kubectl get -f go-demo-2-scaled.yml

kubectl scale deployment go-demo-2-api --replicas 8 --record

kubectl get -f go-demo-2.yml

k3d cluster delete mycluster –all

# Comparison with Docker Swarm

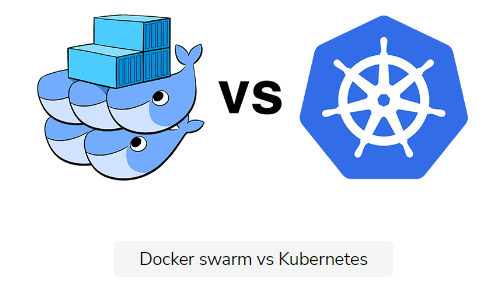
Compare Kubernetes Deployments with Docker Swarm Stacks.

**We'll cover the following**

* [Kubernetes deployments compared to Docker Swarm stacks](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JP8zrr3wN5P#Kubernetes-deployments-compared-to-Docker-Swarm-stacks)
* [Comparing definitions](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JP8zrr3wN5P#Comparing-definitions)
  + [Kubernetes](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JP8zrr3wN5P#Kubernetes)
  + [Docker Swarm](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JP8zrr3wN5P#Docker-Swarm)
* [Comparing functionality](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JP8zrr3wN5P#Comparing-functionality)
  + [Creating objects](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JP8zrr3wN5P#Creating-objects)
  + [Deployment status](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JP8zrr3wN5P#Deployment-status)
  + [Rolling back](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JP8zrr3wN5P#Rolling-back)
  + [Updating multiple deployments](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JP8zrr3wN5P#Updating-multiple-deployments)
* [Conclusion](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JP8zrr3wN5P#Conclusion)

## Kubernetes deployments compared to Docker Swarm stacks

If you have already used Docker Swarm, the logic behind Kubernetes Deployments should be familiar. Both serve the same purpose and can be used to deploy new applications or update those that are already running inside a cluster. In both cases, we can easily deploy new releases without any downtime (when application architecture permits that).



## Comparing definitions

Let’s have a look at the way we define objects in both Kubernetes and Docker Swarm.

### Kubernetes

An example Kubernetes Deployment and Service definition is as follows.

apiVersion: apps/v1

kind: Deployment

metadata:

name: go-demo-2-db

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

ports:

- containerPort: 27017

---

apiVersion: v1

kind: Service

metadata:

name: go-demo-2-db

spec:

ports:

- port: 27017

selector:

type: db

service: go-demo-2

---

apiVersion: apps/v1beta2

kind: Deployment

metadata:

name: go-demo-2-api

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

---

apiVersion: v1

kind: Service

metadata:

name: go-demo-2-api

spec:

type: NodePort

ports:

- port: 8080

selector:

type: api

service: go-demo-2

### Docker Swarm

An equivalent Docker Swarm stack definition is as follows.

version: "3"

services:

api:

image: vfarcic/go-demo-2

environment:

- DB=db

ports:

- 8080

deploy:

replicas: 3

db:

image: mongo:3.3

Both definitions provide, more or less, the same functionality.

It is evident that a Kubernetes Deployment requires a much longer definition with more a complex syntax. It is worth noting that Swarm’s equivalent to readinessProbe and livenessProbe is not present in the stack because it is defined as a HEALTHCHECK inside the Dockerfile. Still, even if we remove them, a Kubernetes Deployment remains longer and more complicated.

When comparing only the differences in the ways to define objects, Docker Swarm is a clear **winner**.

## Comparing functionality

Let’s see what we can conclude from the functional point of view.

### Creating objects

Creating the objects is reasonably straight-forward. Both kubectl create and docker stack deploy will deploy new releases without any downtime. New containers or, in case of Kubernetes, Pods will be created and, in parallel, the old ones will be terminated. So far, both solutions are, more or less, the same.

### Deployment status

One of the **main differences** is what happens in case of a failure. A Kubernetes Deployment will not perform any corrective action in case of a failure. It will stop the update, leaving a combination of new and old Pods running in parallel. Docker Swarm, on the other hand, can be configured to rollback automatically. That might seem like another win for Docker Swarm.

However, Kubernetes has something Swarm doesn’t. We can use kubectl rollout status command to find out whether the update was a success or failure and, in case of the latter, we can undo the rollout. Even though we need a few commands to accomplish the same result, that might fare better when updates are automated. Knowing whether an update succeeded or failed allows us to not only execute a subsequent rollback action but also notify someone that there is a problem.

Both approaches have their pros and cons. Docker Swarm’s automated rollback is better suited in some cases, and Kubernetes update status works better in others. The methods are different, and there is no clear winner, so we’ll proclaim it a **tie**.

### Rolling back

Kubernetes Deployments can record history. We can use the kubectl rollout history command to inspect past rollout. When updates are working as expected, history is not very useful. But, when things go wrong, it might provide additional insight. That can be combined with the ability to rollback to a specific revision, not necessarily the previous one. However, most of the time, we rollback to the previous version. The ability to go back further in time is not very useful. Even when such a need arises, both products can do that.

The **difference** is that Kubernetes Deployments allow us to go to a specific revision (e.g., we’re on the revision five, rollback to the revision two). With Docker Swarm, we’d have to issue a new update (e.g., update the image to the tag 2.0). Since containers are immutable, the result is the same, so the difference is only in the syntax behind a rollback.

The ability to rollback to a specific version or a tag exists in both products. We can argue which syntax is more straightforward or more useful. The differences are minor and we’ll proclaim that there is no winner for that functionality. It’s another **tie**.

### Updating multiple deployments

Since almost everything in Kubernetes is based on label selectors, it has a feature that Docker Swarm doesn’t. We can update multiple Deployments at the same time. We can, for example, issue an update (kubetl set image) that uses filters to find all Mongo databases and upgrade them to a newer release. It is a feature that would require a few lines of bash scripting with Docker Swarm.

However, while the ability to update all Deployments that match specific labels might sound like a useful feature, it often isn’t. More often than not, such actions can produce undesirable effects. If, for example, we have five back-end applications that use Mongo database (one for each), we’d probably want to upgrade them in a more controlled fashion. Teams behind those services would probably want to test each of those upgrades and give their blessings. We probably wouldn’t wait until all are finished, but upgrade a single database when the team in charge of it feels confident.

There are the cases when updating multiple objects is useful so we must give this one to Kubernetes. It is a minor **win**, but it still counts.

## Conclusion

Don’t make a decision based on the differences between Kubernetes Deployments and Docker Swarm stacks.

Definition syntax is where Swarm has a clear win, while on the functional front Kubernetes has a tiny edge over Swarm.

In any case, Kubernetes has much more to offer, and any conclusion based on such a limited comparison scope is bound to be incomplete. We only scratched the surface. Stay tuned for more.

# What's Next?

Recap what we have learned so far in this section and what we are going to learn next.

**We'll cover the following**

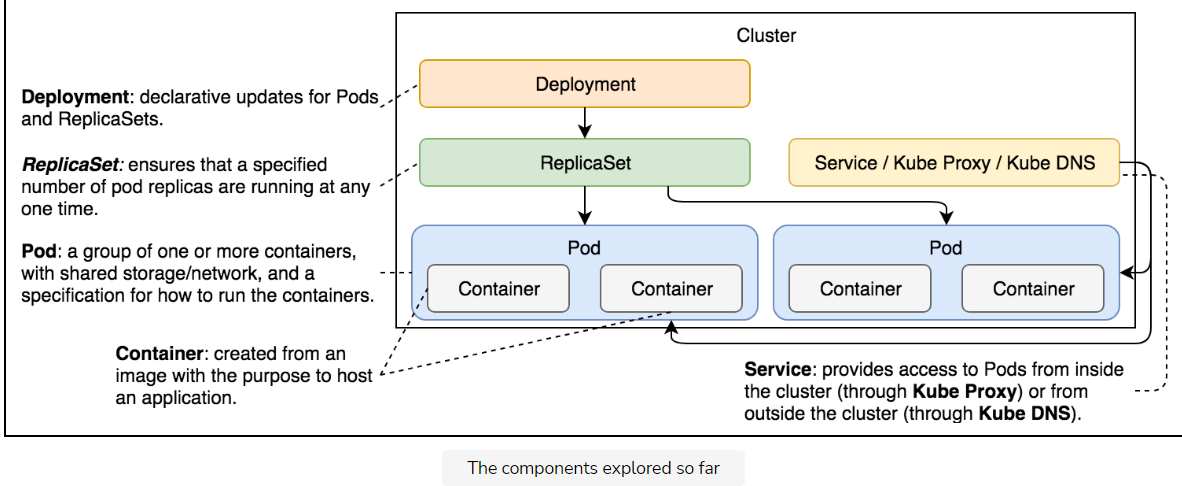
* [Summary](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/YVJpMnDnoYY#Summary)

## Summary

Everything we learned led us to Deployments. Pods must not be created directly, but through ReplicaSets which, similarly, must not be created directly, but through Deployments. They are the objects that allow us not only to create the ReplicaSets and Pods, but that can also be updated without producing any downtime (when applications are designed accordingly).

We combined Deployments with Services so that Pods can communicate with each other, or can be accessed from outside a cluster. All in all, we have everything we need to release our services to production. That is not to say that we understand all the crucial aspects of Kubernetes. We’re not even close to that point. But, we do have almost everything we need for running some types of applications in production. What we’re missing is networking.

ℹ️ If you’d like to know more about Deployments, please explore [Deployment v1 apps](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.24/#deployment-v1-apps) API documentation.



The above illustration shows and defines the Kubernetes components we have explored so far.