**Projects**

* [Projects](https://darey.io/project-listings/)
* [Docs](https://www.darey.io/projects/)
* [(STEP 36): PROJECT 22: Deploying Applications Into Kubernetes Cluster](https://www.darey.io/docs/deploying-applications-into-kubernetes-cluster/)

**DEPLOYING APPLICATIONS INTO KUBERNETES CLUSTER**

[Start Project](https://www.darey.io/docs/deploying-applications-into-kubernetes-cluster/?checkread=1)Submit Project for Review

In this project, you will build upon your knowledge of Kubernetes architecture, and begin to deploy applications on a K8s cluster. Kubernetes has a lot of moving parts; it operates with several layers of abstraction between your application and host machines where it runs. So many terms, and capabilities that is not realistic to learn it all at once. Hence, you will be introduced to as many concepts as possible, but gradually.

Within this project we are going to learn and see in action following:

1. Deployment of software applications using [YAML](https://en.wikipedia.org/wiki/YAML) manifest files with following K8s objects:
   * Pods
   * ReplicaSets
   * Deployments
   * StatefulSets
   * Services (ClusterIP, NodeIP, Loadbalancer)
   * Configmaps
   * Volumes
   * PersistentVolumes
   * PersistentVolumeClaims

…and many more

1. Difference between **stateful** and **stateless** applications
   * Deploy MySQL as a StatefulSet and explain why
2. Limitations of using manifests directly to deploy on K8s
   * Working with [Helm](https://helm.sh/) templates, its components and the most important parts – semantic versioning
   * Converting all the .yaml templates into a helm chart
3. Deploying more tools with Helm charts on AWS Elastic [Kubernetes Service (EKS)](https://aws.amazon.com/eks/)
   * Jenkins  
     -MySQL  
     -Ingress Controllers (Nginx)
   * Cert-Manager
   * Ingress for Jenkins
   * Ingress for the actual application
4. Deploy Monitoring Tools
   * Prometheus
   * Grafana
5. Hybrid CI/CD by combining different tools such as: [Gitlab CICD](https://docs.gitlab.com/ee/ci/), Jenkins. And, you will also be introduced to concepts around [GitOps](https://www.weave.works/technologies/gitops/) using [Weaveworks Flux](https://www.weave.works/oss/flux/).

Instructions On How To Submit Your Work For Review And Feedback

To submit your work for review and feedback – follow [**this instruction**](https://starter-pbl.darey.io/en/latest/submission.html).

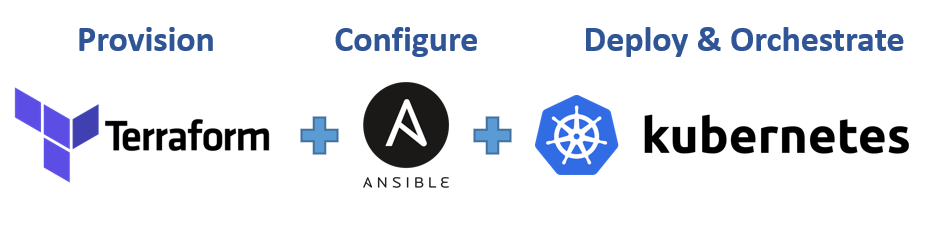
**Choosing the right Kubernetes cluster set up**

When it comes to using a Kubernetes cluster, there is a number of options available depending on the ultimate use of it. For example, if you just need a cluster for development or learning, you can use lightweight tools like [Minikube](https://minikube.sigs.kubernetes.io/docs/start/), or [k3s](https://k3s.io/). These tools can run on your workstation without heavy system requirements. Obviously, there is limit to the amount of workload you can deploy there for testing purposes, but it works exactly like any other Kubernetes cluster.

On the other hand, if you need something more robust, suitable for a production workload and with more advanced capabilities such as horizontal scaling of the worker nodes, then you can consider building own Kubernetes cluster from scratch just as you did in [Project 21](https://expert-pbl.darey.io/en/latest/project21.html). If you have been able to automate the entire bootstrap using Ansible, you can easily spin up your nodes with Terraform, and configure the cluster with your Ansible configuration scripts.

It it a great combination of tools responsible for different parts of your applications:

* **Terraform** for infrastructure provisioning
* **Ansible** for cluster master and worker nodes configuration
* **Kubernetes** for deploying your containerized application and orchestrating the deployment



Other options will be to leverage a [Managed Service](https://www.adept.co.uk/the-benefits-of-cloud-managed-services-for-business/) Kubernetes cluster from public cloud providers such as: [AWS EKS](https://aws.amazon.com/eks), [Microsoft AKS](https://azure.microsoft.com/en-gb/services/kubernetes-service), or [Google Cloud Platform GKE](https://cloud.google.com/kubernetes-engine). There are so many more options out there. Regardless of whichever one you choose, the experience is usually very similar.

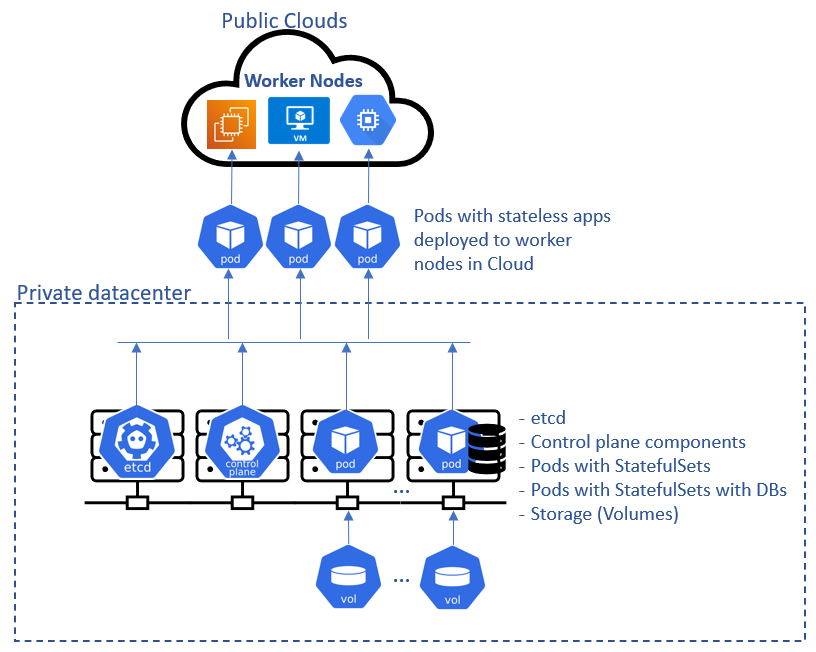
Most organisations choose Managed Service options for obvious reasons such as:

1. Less administrative overheads
2. Reduced cost of ownership
3. Improved Security
4. Seamless support
5. Periodical updates to a stable and well-tested version
6. Faster cluster spin up

… and many more

However, there is usually strong reasons why organisations with very strict compliance and security concerns choose to build their own Kubernetes clusters. Most of the companies that go this route will mostly use on-premises data centres. When there is need to store data privately due to its sensitive nature, companies will rather not use a public cloud provider. Because, if they do, they have no idea of the physical location of the data centre in which their data is being persisted. Banks and Governments are typical examples of this.

Some setup options can combine both public and private cloud together. For example, the master nodes, etcd clusters, and some worker nodes that run [stateful](https://whatis.techtarget.com/definition/stateful-app) applications can be configured in private datacentres, while worker nodes that require heavy computations and [stateless](https://www.redhat.com/en/topics/cloud-native-apps/stateful-vs-stateless) applications can run in public clouds. This kind of hybrid architecture is ideal to satisfy compliance, while also benefiting from other public cloud capabilities.



**Deploying the Tooling app using Kubernetes objects**

In this section, you will begin to write configuration files for Kubernetes objects (they are usually referred as manifests) in the form of files with yaml syntax and deploy them using kubectl console. But first, let us understand what a Kubernetes object is.

**Kubernetes objects** are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of your cluster. Specifically, they can describe:

* What containerized applications are running (and on which nodes)
* The resources available to those applications
* The policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance

These objects are ***"record of intent"*** – once you create the object, the Kubernetes system will constantly work to ensure that the object exists. By creating an object, you are effectively telling the Kubernetes system what you want your cluster’s workload to look like; this is your cluster’s desired state.

To work with Kubernetes objects – whether to create, modify, or delete them – you will need to use the Kubernetes API. When you use the kubectl command-line interface, for example, the CLI makes the necessary Kubernetes API calls for you. It is also possible to use curl to directly interact with the Kubernetes API, or it can be as part of developing a program in different programming languages. That will require some advance knowledge. You can [read more about client libraries](https://kubernetes.io/docs/reference/using-api/client-libraries/) to get an idea on how that works.

# UNDERSTANDING THE CONCEPT

Let us try to understand a bit more about how the service object is able to route traffic to the Pod.

If you run the below command:

kubectl get service nginx-service -o wide

You will get the output similar to this:

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

nginx-service ClusterIP 10.100.71.130 <none> 80/TCP 4d app=nginx-pod

As you already know, the service’s type is ClusterIP, and in the above output, it has the IP address of 10.100.71.130 – This IP works just like an internal loadbalancer. It accepts requests and forwards it to an IP address of any Pod that has the respective selector label. In this case, it is app=nginx-pod. If there is more than one Pod with that label, service will distribute the traffic to all these pods in a [Round Robin](https://en.wikipedia.org/wiki/Round-robin_scheduling) fashion.

Now, let us have a look at what the Pod looks like:

kubectl get pod nginx-pod --show-labels

**Output:**

NAME READY STATUS RESTARTS AGE LABELS

nginx-pod 1/1 Running 0 31m app=nginx-pod

**Notice that the IP address of the Pod, is *NOT* the IP address of the server it is running on. Kubernetes, through the implementation of network plugins assigns virtual IP adrresses to each Pod.**

kubectl get pod nginx-pod -o wide

**Output:**

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES

nginx-pod 1/1 Running 0 57m 172.50.197.236 ip-172-50-197-215.eu-central-1.compute.internal <none> <none>

Therefore, Service with IP 10.100.71.130 takes request and forwards to Pod with IP 172.50.197.236

#### Self Side Task:

1. Build the Tooling app Dockerfile and push it to Dockerhub registry
2. Write a Pod and a Service manifests, ensure that you can access the Tooling app’s frontend using port-forwarding feature.

#### Expose a Service on a server’s public IP address & static port

Sometimes, it may be needed to directly access the application using the public IP of the server (when we speak of a K8s cluster we can replace ‘server’ with ‘node’) the Pod is running on. This is when the [**NodePort**](https://kubernetes.io/docs/concepts/services-networking/service/#nodeport) service type comes in handy.

A **Node port** service type exposes the service on a static port on the node’s IP address. NodePorts are in the 30000-32767 range by default, which means a NodePort is unlikely to match a service’s intended port (for example, 80 may be exposed as 30080).

Update the nginx-service yaml to use a NodePort Service.

apiVersion: v1

kind: Service

metadata:

name: nginx-service

spec:

type: NodePort

selector:

app: nginx-pod

ports:

- protocol: TCP

port: 80

nodePort: 30080

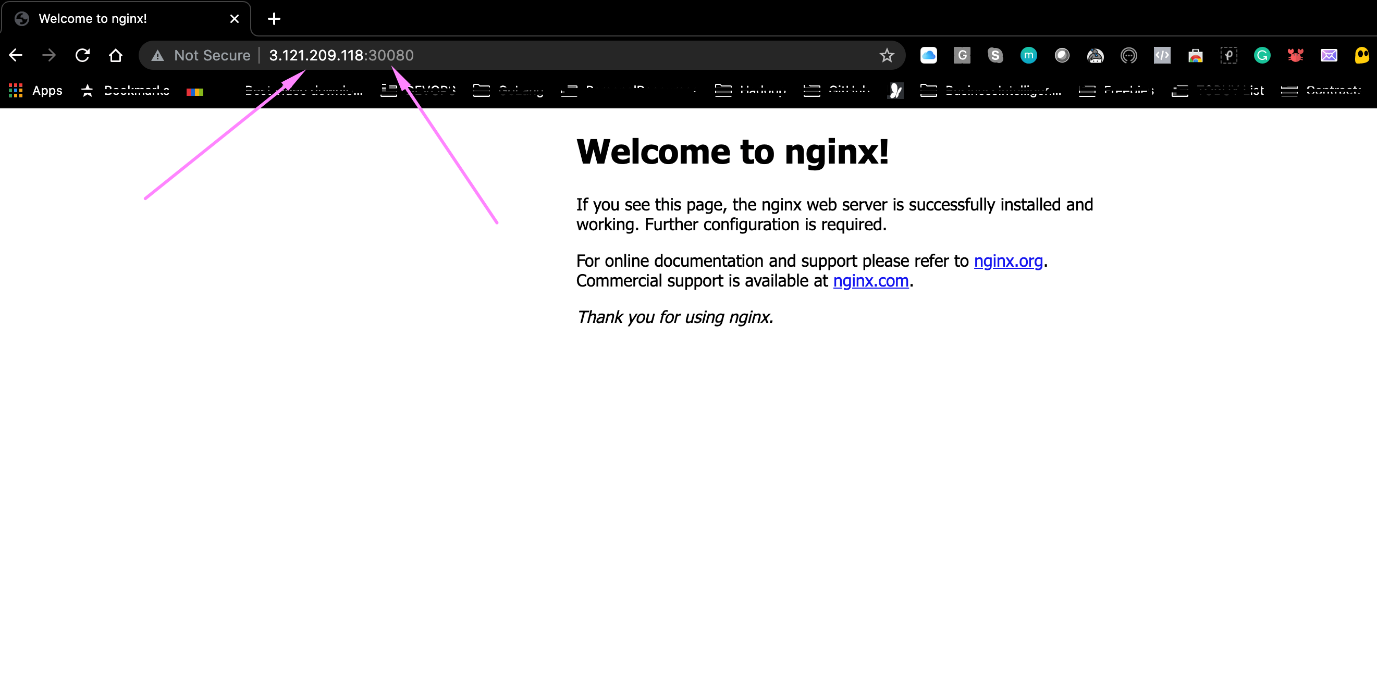
What has changed is:

1. Specified the type of service (Nodeport)
2. Specified the NodePort number to use.

To access the service, you must:

* Allow the inbound traffic in your EC2’s Security Group to the NodePort range 30000-32767
* Get the public IP address of the node the Pod is running on, append the nodeport and access the app through the browser.

You must understand that the port number 30080 is a port on the node in which the Pod is scheduled to run. If the Pod ever gets rescheduled elsewhere, that the same port number will be used on the new node it is running on. So, if you have multiple Pods running on several nodes at the same time – they all will be exposed on respective nodes’ IP addresses with a static port number.



Read some more information regarding Services in Kubernetes in [this article](https://medium.com/avmconsulting-blog/service-types-in-kubernetes-24a1587677d6).

#### How Kubernetes ensures desired number of Pods is always running?

When we define a Pod manifest and appy it – we create a Pod that is running until it’s terminated for some reason (e.g., error, Node reboot or some other reason), but what if we want to declare that we always need at least 3 replicas of the same Pod running at all times? Then we must use an [**ResplicaSet (RS)**](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/) object – it’s purpose is to maintain a stable set of Pod replicas running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods.

**Note:** In some older books or documents you might find the old version of a similar object – [ReplicationController (RC)](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/), it had similar purpose, but did not support [set-base label selectors](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#set-based-requirement) and it is now recommended to use ReplicaSets instead, since it is the next-generation RC.

Let us delete our nginx-pod Pod:

kubectl delete -f nginx-pod.yaml

**Output:**

pod "nginx-pod" deleted

# COMMON KUBERNETES OBJECTS

* Pod
* Namespace
* ResplicaSet (Manages Pods)
* DeploymentController (Manages Pods)
* StatefulSet
* DaemonSet
* Service
* ConfigMap
* Volume
* Job/Cronjob

The very first concept to understand is the difference between how **Docker** and **Kubernetes** run containers – with Docker, every docker run command will run an image (representing an application) as a container. The running container is a Docker’s smallest entity, it is the most basic deployable object. Kubernetes on the other hand operates with pods instead of containers, a pods encapsulates a container. Kubernetes uses pods as its smallest, and most basic deployable object with a unique feature that allows it to run multiple containers within a single Pod. It is not the most common pattern – to have more than one container in a Pod, but there are cases when this capability comes in handy.

In the world of docker, or docker compose, to run the Tooling app, you must deploy separate containers for the application and the database. But in the world of Kubernetes, you can run both: application and database containers in the same Pod. When multiple containers run within the same Pod, they can directly communicate with each other as if they were running on the same localhost. Although running both the application and database in the same Pod is **NOT** a recommended approach.

A Pod that contains one container is called single container pod and it is the most common Kubernetes use case. A Pod that contains multiple co-related containers is called multi-container pod. There are few patterns for multi-container Pods; one of them is the [**sidecar**](https://medium.com/bb-tutorials-and-thoughts/kubernetes-learn-sidecar-container-pattern-6d8c21f873d) container pattern – it means that in the same Pod there is a main container and an auxiliary one that extends and enhances the functionality of the main one without changing it.

There are other patterns, such as: [init container](https://medium.com/bb-tutorials-and-thoughts/kubernetes-learn-init-container-pattern-7a757742de6b), [adapter container](https://medium.com/bb-tutorials-and-thoughts/kubernetes-learn-adaptor-container-pattern-97674285983c), [ambassador container](https://medium.com/bb-tutorials-and-thoughts/kubernetes-learn-ambassador-container-pattern-bc2e1331bd3a). These are more advanced topics that you can study on your own, let us continue with the other objects.

We will not go into the theoretical details of all the objects, rather we will begin to experience them in action.

#### Understanding the common YAML fields for every Kubernetes object

Every Kubernetes object includes object fields that govern the object’s configuration:

* **kind**: Represents the type of kubernetes object created. It can be a Pod, DaemonSet, Deployments or Service.
* **version**: Kubernetes api version used to create the resource, it can be v1, v1beta and v2. Some of the kubernetes features can be released under beta and available for general public usage.
* **metadata**: provides information about the resource like name of the Pod, namespace under which the Pod will be running,  
  labels and annotations.
* **spec**: consists of the core information about Pod. Here we will tell kubernetes what would be the expected state of resource, Like container image, number of replicas, environment variables and volumes.
* **status**: consists of information about the running object, status of each container. Status field is supplied and updated by Kubernetes after creation. This is not something you will have to put in the YAML manifest.

#### Deploying a random Pod

Lets see what it looks like to have a Pod running in a k8s cluster. This section is just to illustrate and get you to familiarise with how the object’s fields work. Lets deploy a basic Nginx container to run inside a Pod.

* **apiVersion** is **v1**
* **kind** is **Pod**
* **metatdata** has a name which is set to **nginx-pod**
* The **spec** section has further information about the Pod. Where to find the image to run the container – (This defaults to **Docker Hub**), the port and protocol.

The structure is similar for any Kubernetes objects, and you will get to see them all as we progress.

1. Create [a Pod](https://kubernetes.io/docs/concepts/workloads/pods/) yaml manifest on your master node
2. sudo cat <<EOF | sudo tee ./nginx-pod.yaml
3. apiVersion: v1
4. kind: Pod
5. metadata:
6. name: nginx-pod
7. spec:
8. containers:
9. - image: nginx:latest
10. name: nginx-pod
11. ports:
12. - containerPort: 80
13. protocol: TCP

EOF

1. Apply the manifest with the help of kubectl

kubectl apply -f nginx-pod.yaml

**Output:**

pod/nginx-pod created

1. Get an output of the pods running in the cluster

kubectl get pods

**Output:**

NAME READY STATUS RESTARTS AGE

nginx-pod 1/1 Running 0 19m

1. If the Pods were not ready for any reason, for example if there are no worker nodes, you will see something like the below output.

NAME READY STATUS RESTARTS AGE

nginx-pod 0/1 Pending 0 111s

1. To see other fields introduced by kubernetes after you have deployed the resource, simply run below command, and examine the output. You will see other fields that kubernetes updates from time to time to represent the state of the resource within the cluster. -o simply means the **output** format.

kubectl get pod nginx-pod -o yaml

or

kubectl describe pod nginx-pod

# ACCESSING THE APP FROM THE BROWSER

Now you have a running Pod. What’s next?

The ultimate goal of any solution is to access it either through a web portal or some application (e.g., mobile app). We have a Pod with Nginx container, so we need to access it from the browser. But all you have is a running Pod that has its own IP address which cannot be accessed through the browser. To achieve this, we need another Kubernetes object called [**Service**](https://kubernetes.io/docs/concepts/services-networking/service/) to accept our request and pass it on to the Pod.

A service is an object that accepts requests on behalf of the Pods and forwards it to the Pod’s IP address. If you run the command below, you will be able to see the Pod’s IP address. But there is no way to reach it directly from the outside world.

kubectl get pod nginx-pod -o wide

**Output:**

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES

nginx-pod 1/1 Running 0 138m 172.50.202.214 ip-172-50-202-161.eu-central-1.compute.internal <none> <none>

Let us try to access the Pod through its IP address from within the K8s cluster. To do this,

1. We need an image that already has curl software installed. You can check it out [here](https://hub.docker.com/r/dareyregistry/curl)

dareyregistry/curl

1. Run kubectl to connect inside the container

kubectl run curl --image=dareyregistry/curl -i --tty

1. Run curl and point to the IP address of the Nginx Pod (Use the IP address of your own Pod)

# curl -v 172.50.202.214:80

**Output:**

> GET / HTTP/1.1

> User-Agent: curl/7.35.0

> Host: 172.50.202.214

> Accept: \*/\*

>

< HTTP/1.1 200 OK

< Server: nginx/1.21.0

< Date: Sat, 12 Jun 2021 21:12:56 GMT

< Content-Type: text/html

< Content-Length: 612

< Last-Modified: Tue, 25 May 2021 12:28:56 GMT

< Connection: keep-alive

< ETag: "60aced88-264"

< Accept-Ranges: bytes

<

<!DOCTYPE html>

<html>

<head>

<title>Welcome to nginx!</title>

<style>

body {

width: 35em;

margin: 0 auto;

font-family: Tahoma, Verdana, Arial, sans-serif;

}

</style>

</head>

<body>

<h1>Welcome to nginx!</h1>

<p>If you see this page, the nginx web server is successfully installed and

working. Further configuration is required.</p>

<p>For online documentation and support please refer to

<a href="http://nginx.org/">nginx.org</a>.<br/>

Commercial support is available at

<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>

</body>

</html>

If the use case for your solution is required for internal use ONLY, without public Internet requirement. Then, this should be OK. But in most cases, it is NOT!

Assuming that your requirement is to access the Nginx Pod internally, using the Pod’s IP address directly as above is not a reliable choice because Pods are ephemeral. They are not designed to run forever. When they die and another Pod is brought back up, the IP address will change and any application that is using the previous IP address directly will break.

To solve this problem, kubernetes uses **Service** – An object that abstracts the underlining IP addresses of Pods. A service can serve as a load balancer, and a reverse proxy which basically takes the request using a human readable DNS name, resolves to a Pod IP that is running and forwards the request to it. This way, you do not need to use an IP address. Rather, you can simply refer to the service name directly.

Let us create a service to access the **Nginx Pod**

1. Create a Service yaml manifest file:

sudo cat <<EOF | sudo tee ./nginx-service.yaml

apiVersion: v1

kind: Service

metadata:

name: nginx-service

spec:

selector:

app: nginx-pod

ports:

- protocol: TCP

port: 80

targetPort: 80

EOF

1. Create a nginx-service resource by applying your manifest

kubectl apply -f nginx-service.yaml

**output:**

service/nginx-service created

1. Check the created service

kubectl get service

**output:**

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

kubernetes ClusterIP 10.100.0.1 <none> 443/TCP 68d

nginx-service ClusterIP 10.100.71.130 <none> 80/TCP 85s

**Observation:**

The **TYPE** column in the output shows that there are [different service types](https://kubernetes.io/docs/concepts/services-networking/service/#publishing-services-service-types).

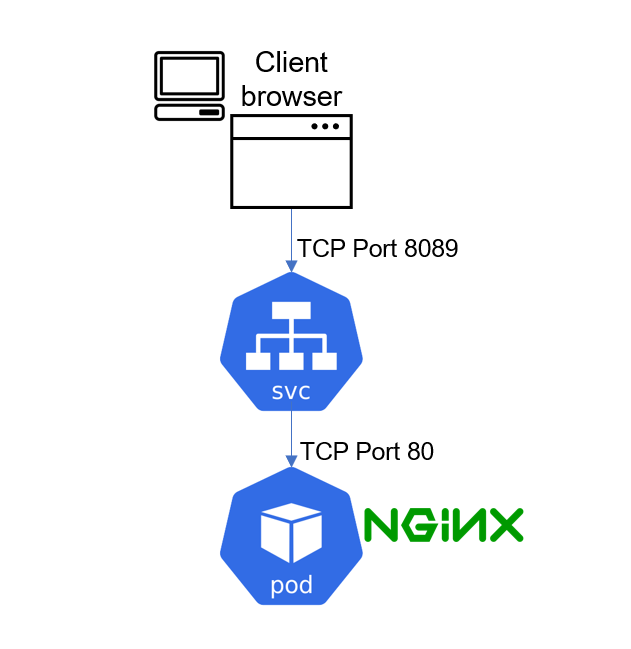
* ClusterIP
* NodePort
* LoadBalancer &
* Headless Service

Since we did not specify any type, it is obvious that the default type is **ClusterIP**

Now that we have a service created, how can we access the app? Since there is no public IP address, we can leverage kubectl's **port-forward** functionality.

kubectl port-forward svc/nginx-service 8089:80

**8089** is an arbitrary port number on your laptop or client PC, and we want to tunnel traffic through it to the port number of the nginx-service **80**.



Unfortunately, this will not work quite yet. Because there is no way the service will be able to select the actual Pod it is meant to route traffic to. If there are hundreds of Pods running, there must be a way to ensure that the service only forwards requests to the specific Pod it is intended for.

To make this work, you must reconfigure the Pod manifest and introduce **labels** to match the **selectors** key in the field section of the service manifest.

1. Update the Pod manifest with the below and apply the manifest:

apiVersion: v1

kind: Pod

metadata:

name: nginx-pod

labels:

app: nginx-pod

spec:

containers:

- image: nginx:latest

name: nginx-pod

ports:

- containerPort: 80

protocol: TCP

Notice that under the metadata section, we have now introduced labels with a key field called app and its value nginx-pod. This matches exactly the selector key in the **service** manifest.

The key/value pairs can be anything you specify. These are not Kubernetes specific keywords. As long as it matches the selector, the service object will be able to route traffic to the Pod.

Apply the manifest with:

kubectl apply -f nginx-pod.yaml

1. Run kubectl port-forward command again

kubectl port-forward svc/nginx-service 8089:80

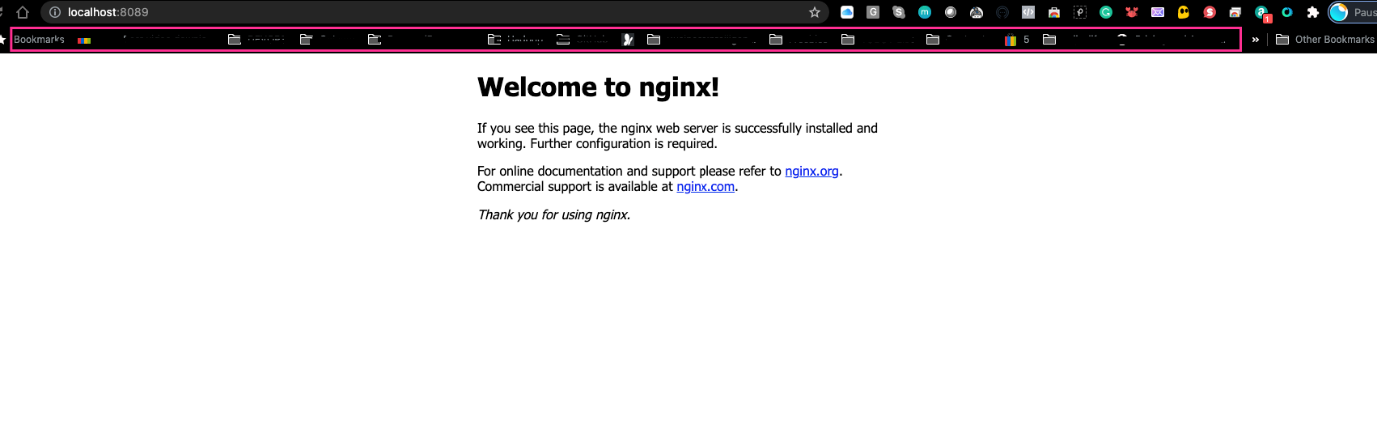
**output:**

kubectl port-forward svc/nginx-service 8089:80

Forwarding from 127.0.0.1:8089 -> 80

Forwarding from [::1]:8089 -> 80

Then go to your web browser and enter localhost:8089 – You should now be able to see the nginx page in the browser.



# CREATE A REPLICA SET

Let us create a rs.yaml manifest for a ReplicaSet object:

#Part 1

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: nginx-rs

spec:

replicas: 3

selector:

app: nginx-pod

#Part 2

template:

metadata:

name: nginx-pod

labels:

app: nginx-pod

spec:

containers:

- image: nginx:latest

name: nginx-pod

ports:

- containerPort: 80

protocol: TCP

kubectl apply -f rs.yaml

The manifest file of ReplicaSet consist of the following fields:

* apiVersion: This field specifies the version of kubernetes Api to which the object belongs. ReplicaSet belongs to **apps/v1** apiVersion.
* kind: This field specify the type of object for which the manifest belongs to. Here, it is **ReplicaSet**.
* metadata: This field includes the metadata for the object. It mainly includes two fields: name and labels of the ReplicaSet.
* spec: This field specifies the **label selector** to be used to select the Pods, number of replicas of the Pod to be run and the container or list of containers which the Pod will run. In the above example, we are running 3 replicas of nginx container.

Let us check what Pods have been created:

kubectl get pods

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES

nginx-pod-j784r 1/1 Running 0 7m41s 172.50.197.5 ip-172-50-197-52.eu-central-1.compute.internal <none> <none>

nginx-pod-kg7v6 1/1 Running 0 7m41s 172.50.192.152 ip-172-50-192-173.eu-central-1.compute.internal <none> <none>

nginx-pod-ntbn4 1/1 Running 0 7m41s 172.50.202.162 ip-172-50-202-18.eu-central-1.compute.internal <none> <none>

Here we see three ngix-pods with some random suffixes (e.g., -j784r) – it means, that these Pods were created and named automatically by some other object (higher level of abstraction) such as ReplicaSet.

Try to delete one of the Pods:

kubectl delete po nginx-pod-j784r

**Output:**

pod "nginx-pod-j784r" deleted

❯ kubectl get pods

NAME READY STATUS RESTARTS AGE

nginx-rc-7xt8z 1/1 Running 0 22s

nginx-rc-kg7v6 1/1 Running 0 34m

nginx-rc-ntbn4 1/1 Running 0 34m

You can see, that we still have all 3 Pods, but one has been recreated (can you differentiate the new one?)

Explore the ReplicaSet created:

kubectl get rs -o wide

**Output:**

NAME DESIRED CURRENT READY AGE CONTAINERS IMAGES SELECTOR

nginx-rs 3 3 3 34m nginx-pod nginx:latest app=nginx-pod

Notice, that ReplicaSet understands which Pods to create by using **SELECTOR** key-value pair.

##### Get detailed information of a ReplicaSet

To display detailed information about any Kubernetes object, you can use 2 differen commands:

* kubectl **describe** %object\_type% %object\_name% (e.g. kubectl describe rs nginx-rs)
* kubectl **get** %object\_type% %object\_name% -o yaml (e.g. kubectl describe rs nginx-rs -o yaml)

Try both commands in action and see the difference. Also try **get** with -o json instead of -o yaml and decide for yourself which output option is more readable for you.

##### Scale ReplicaSet up and down:

In general, there are 2 approaches of [Kubernetes Object Management](https://kubernetes.io/docs/concepts/overview/working-with-objects/object-management/): imperative and declarative.

Let us see how we can use both to scale our Replicaset up and down:

**Imperative:**

We can easily scale our ReplicaSet up by specifying the desired number of replicas in an imperative command, like this:

❯ kubectl scale rs nginx-rs --replicas=5

replicationcontroller/nginx-rc scaled

❯ kubectl get pods

NAME READY STATUS RESTARTS AGE

nginx-rc-4kgpj 1/1 Running 0 4m30s

nginx-rc-4z2pn 1/1 Running 0 4m30s

nginx-rc-g4tvg 1/1 Running 0 6s

nginx-rc-kmh8m 1/1 Running 0 6s

nginx-rc-zlgvp 1/1 Running 0 4m30s

Scaling down will work the same way, so scale it down to 3 replicas.

**Declarative:**

Declarative way would be to open our rs.yaml manifest, change desired number of replicas in respective section

spec:

replicas: 3

and applying the updated manifest:

kubectl apply -f rs.yaml

There is another method – **‘ad-hoc’**, it is definitely not the best practice and we do not recommend using it, but you can edit an existing ReplicaSet with following command:

kubectl edit -f rs.yaml

##### Advanced label matching

As Kubernetes mature as a technology, so does its features and improvements to k8s objects. ReplicationControllers do not meet certain complex business requirements when it comes to using selectors. Imagine if you need to select Pods with multiple lables that represents things like:

* **Application tier:** such as Frontend, or Backend
* **Environment:** such as Dev, SIT, QA, Preprod, or Prod

So far, we used a simple selector that just matches a key-value pair and check only ‘equality’:

selector:

app: nginx-pod

But in some cases, we want ReplicaSet to manage our existing containers that match certain criteria, we can use the same simple label matching or we can use some more complex conditions, such as:

- in

- not in

- not equal

- etc...

Let us look at the following manifest file:

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: nginx-rs

spec:

replicas: 3

selector:

matchLabels:

env: prod

matchExpressions:

- { key: tier, operator: In, values: [frontend] }

template:

metadata:

name: nginx

labels:

env: prod

tier: frontend

spec:

containers:

- name: nginx-container

image: nginx:latest

ports:

- containerPort: 80

protocol: TCP

In the above spec file, under the selector, **matchLabels** and **matchExpression** are used to specify the key-value pair. The **matchLabel** works exactly the same way as the equality-based selector, and the matchExpression is used to specify the set based selectors. This feature is the main differentiator between **ReplicaSet** and previously mentioned obsolete **ReplicationController**.

Get the replication set:

❯ kubectl get rs nginx-rs -o wide

NAME DESIRED CURRENT READY AGE CONTAINERS IMAGES SELECTOR

nginx-rs 3 3 3 5m34s nginx-container nginx:latest env=prod,tier in (frontend)

# USING AWS LOAD BALANCER TO ACCESS YOUR SERVICE IN KUBERNETES.

**Note:** You will only be able to test this using AWS EKS. You don not have to set this up in current project yet. In the next project, you will update your Terraform code to build an EKS cluster.

You have previously accessed the Nginx service through **ClusterIP**, and **NodeIP**, but there is another service type – [**Loadbalancer**](https://kubernetes.io/docs/concepts/services-networking/service/#loadbalancer). This type of service does not only create a **Service** object in K8s, but also provisions a real external Load Balancer (e.g. [Elastic Load Balancer – ELB](https://aws.amazon.com/elasticloadbalancing/) in AWS)

To get the experience of this service type, update your service manifest and use the **LoadBalancer** type. Also, ensure that the selector references the Pods in the replica set.

apiVersion: v1

kind: Service

metadata:

name: nginx-service

spec:

type: LoadBalancer

selector:

tier: frontend

ports:

- protocol: TCP

port: 80 # This is the port the Loadbalancer is listening at

targetPort: 80 # This is the port the container is listening at

Apply the configuration:

kubectl apply -f nginx-service.yaml

Get the newly created service :

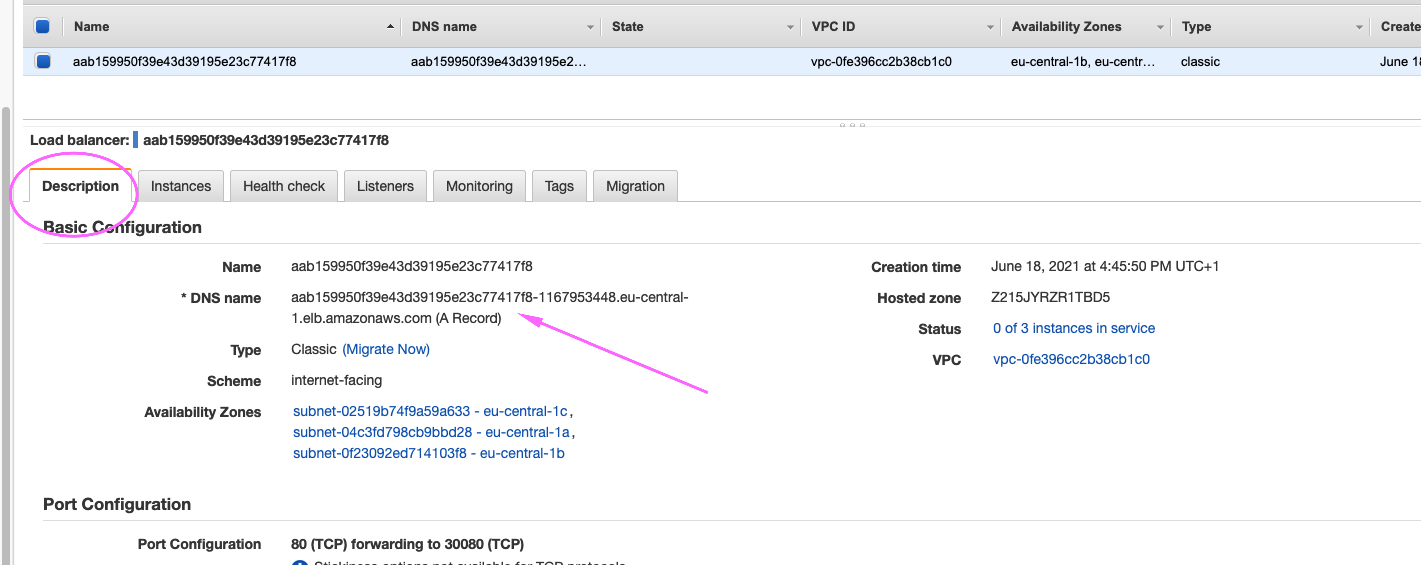
kubectl get service nginx-service

**output:**

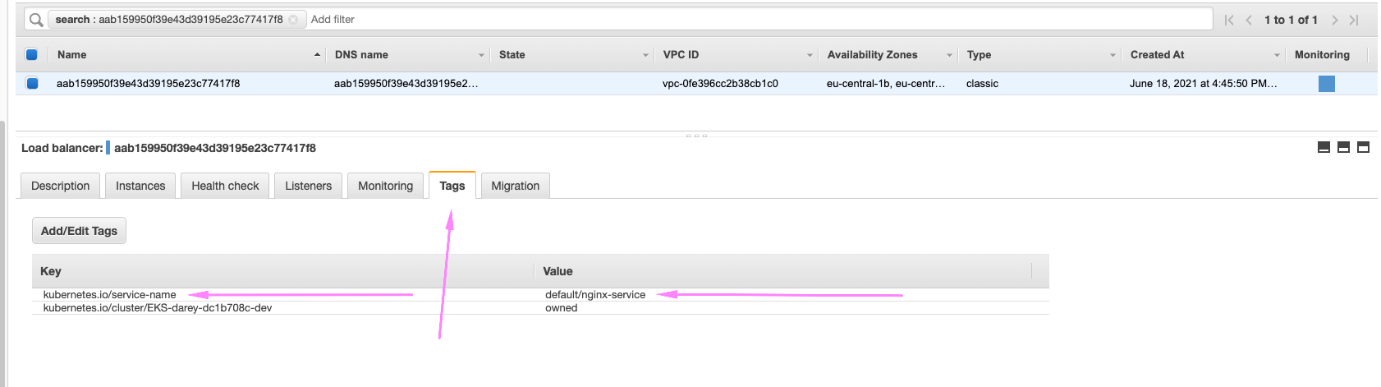
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

nginx-service LoadBalancer 10.100.71.130 aab159950f39e43d39195e23c77417f8-1167953448.eu-central-1.elb.amazonaws.com 80:31388/TCP 5d18h

An ELB resource will be created in your AWS console.



A Kubernetes component in the control plane called [**Cloud-controller-manager**](https://kubernetes.io/docs/concepts/architecture/cloud-controller) is responsible for triggeriong this action. It connects to your specific cloud provider’s (AWS) APIs and create resources such as Load balancers. It will ensure that the resource is appropriately tagged:



Get the output of the entire yaml for the service. You will some additional information about this service in which you did not define them in the yaml manifest. Kubernetes did this for you.

kubectl get service nginx-service -o yaml

**output:**

apiVersion: v1

kind: Service

metadata:

annotations:

kubectl.kubernetes.io/last-applied-configuration: |

{"apiVersion":"v1","kind":"Service","metadata":{"annotations":{},"name":"nginx-service","namespace":"default"},"spec":{"ports":[{"port":80,"protocol":"TCP","targetPort":80}],"selector":{"app":"nginx-pod"},"type":"LoadBalancer"}}

creationTimestamp: "2021-06-18T16:24:21Z"

finalizers:

- service.kubernetes.io/load-balancer-cleanup

name: nginx-service

namespace: default

resourceVersion: "21824260"

selfLink: /api/v1/namespaces/default/services/nginx-service

uid: c12145d6-a8b5-491d-95ff-8e2c6296b46c

spec:

clusterIP: 10.100.153.44

externalTrafficPolicy: Cluster

ports:

- nodePort: 31388

port: 80

protocol: TCP

targetPort: 80

selector:

tier: frontend

sessionAffinity: None

type: LoadBalancer

status:

loadBalancer:

ingress:

- hostname: ac12145d6a8b5491d95ff8e2c6296b46-588706163.eu-central-1.elb.amazonaws.com

1. A clusterIP key is updated in the manifest and assigned an IP address. Even though you have specified a Loadbalancer service type, internally it still requires a clusterIP to route the external traffic through.
2. In the ports section, nodePort is still used. This is because Kubernetes still needs to use a dedicated port on the worker node to route the traffic through. Ensure that port range 30000-32767 is opened in your inbound Security Group configuration.
3. More information about the provisioned balancer is also published in the .status.loadBalancer field.

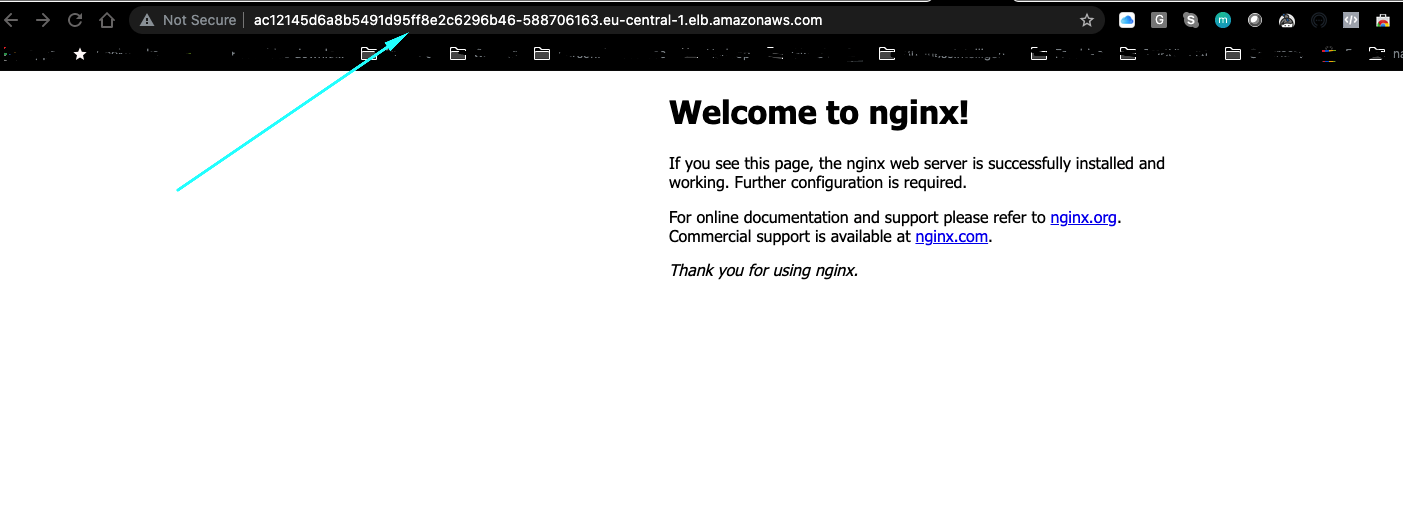
status:

loadBalancer:

ingress:

- hostname: ac12145d6a8b5491d95ff8e2c6296b46-588706163.eu-central-1.elb.amazonaws.com

Copy and paste the load balancer’s address to the browser, and you will access the Nginx service



# USING DEPLOYMENT CONTROLLERS

#### Do not Use Replication Controllers – Use Deployment Controllers Instead

Kubernetes is loaded with a lot of features, and with its vibrant open source community, these features are constantly evolving and adding up.

Previously, you have seen the improvements from **ReplicationControllers (RC)**, to **ReplicaSets (RS)**. In this section you will see another K8s object which is highly recommended over Replication objects (RC and RS).

A [**Deployment**](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) is another layer above ReplicaSets and Pods, newer and more advanced level concept than ReplicaSets. It manages the deployment of ReplicaSets and allows for easy updating of a ReplicaSet as well as the ability to roll back to a previous version of deployment. It is declarative and can be used for rolling updates of micro-services, ensuring there is no downtime.

Officially, it is highly recommended to use **Deplyments** to manage replica sets rather than using replica sets directly.

Let us see Deployment in action.

1. Delete the ReplicaSet

kubectl delete rs nginx-rs

1. Understand the layout of the deployment.yaml manifest below. Lets go through the 3 separated sections:

# Section 1 - This is the part that defines the deployment

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

labels:

tier: frontend

# Section 2 - This is the Replica set layer controlled by the deployment

spec:

replicas: 3

selector:

matchLabels:

tier: frontend

# Section 3 - This is the Pod section controlled by the deployment and selected by the replica set in section 2.

template:

metadata:

labels:

tier: frontend

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

1. Putting them altogether

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

labels:

tier: frontend

spec:

replicas: 3

selector:

matchLabels:

tier: frontend

template:

metadata:

labels:

tier: frontend

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 8

kubectl apply -f deployment.yaml

Run commands to get the following

1. Get the Deployment
2. NAME READY UP-TO-DATE AVAILABLE AGE

nginx-deployment 3/3 3 3 39s

1. Get the ReplicaSet
2. NAME DESIRED CURRENT READY AGE

nginx-deployment-56466d4948 3 3 3 24s

1. Get the Pods
2. NAME READY STATUS RESTARTS AGE
3. nginx-deployment-56466d4948-5zdbx 1/1 Running 0 12s
4. nginx-deployment-56466d4948-tg9j8 1/1 Running 0 12s

nginx-deployment-56466d4948-ttn5t 1/1 Running 0 12s

1. Scale the replicas in the Deployment to 15 Pods
2. NAME READY STATUS RESTARTS AGE
3. nginx-deployment-56466d4948-58nqx 1/1 Running 0 6s
4. nginx-deployment-56466d4948-5z4c2 1/1 Running 0 6s
5. nginx-deployment-56466d4948-5zdbx 1/1 Running 0 17m
6. nginx-deployment-56466d4948-78j9c 1/1 Running 0 6s
7. nginx-deployment-56466d4948-gj4fd 1/1 Running 0 6s
8. nginx-deployment-56466d4948-gsrpz 1/1 Running 0 6s
9. nginx-deployment-56466d4948-kg9hp 1/1 Running 0 6s
10. nginx-deployment-56466d4948-qs29b 1/1 Running 0 6s
11. nginx-deployment-56466d4948-sfft6 1/1 Running 0 6s
12. nginx-deployment-56466d4948-sg4np 1/1 Running 0 6s
13. nginx-deployment-56466d4948-tg9j8 1/1 Running 0 17m
14. nginx-deployment-56466d4948-ttn5t 1/1 Running 0 17m
15. nginx-deployment-56466d4948-vfmjx 1/1 Running 0 6s
16. nginx-deployment-56466d4948-vlgbs 1/1 Running 0 6s

nginx-deployment-56466d4948-xctfh 1/1 Running 0 6s

1. Exec into one of the Pod’s container to run Linux commands

kubectl exec -it nginx-deployment-56466d4948-78j9c bash

List the files and folders in the Nginx directory

root@nginx-deployment-56466d4948-78j9c:/# ls -ltr /etc/nginx/

total 24

-rw-r--r-- 1 root root 664 May 25 12:28 uwsgi\_params

-rw-r--r-- 1 root root 636 May 25 12:28 scgi\_params

-rw-r--r-- 1 root root 5290 May 25 12:28 mime.types

-rw-r--r-- 1 root root 1007 May 25 12:28 fastcgi\_params

-rw-r--r-- 1 root root 648 May 25 13:01 nginx.conf

lrwxrwxrwx 1 root root 22 May 25 13:01 modules -> /usr/lib/nginx/modules

drwxr-xr-x 1 root root 26 Jun 18 22:08 conf.d

Check the content of the default Nginx configuration file

root@nginx-deployment-56466d4948-78j9c:/# cat /etc/nginx/conf.d/default.conf

server {

listen 80;

listen [::]:80;

server\_name localhost;

#access\_log /var/log/nginx/host.access.log main;

location / {

root /usr/share/nginx/html;

index index.html index.htm;

}

#error\_page 404 /404.html;

# redirect server error pages to the static page /50x.html

#

error\_page 500 502 503 504 /50x.html;

location = /50x.html {

root /usr/share/nginx/html;

}

# proxy the PHP scripts to Apache listening on 127.0.0.1:80

#

#location ~ \.php$ {

# proxy\_pass http://127.0.0.1;

#}

# pass the PHP scripts to FastCGI server listening on 127.0.0.1:9000

#

#location ~ \.php$ {

# root html;

# fastcgi\_pass 127.0.0.1:9000;

# fastcgi\_index index.php;

# fastcgi\_param SCRIPT\_FILENAME /scripts$fastcgi\_script\_name;

# include fastcgi\_params;

#}

# deny access to .htaccess files, if Apache's document root

# concurs with nginx's one

#

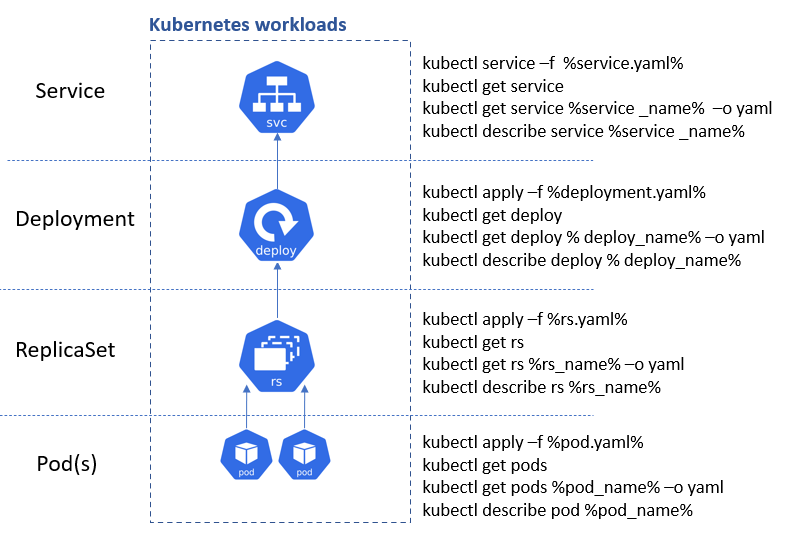
#location ~ /\.ht {

# deny all;

#}

}

Now, as we have got acquaited with most common Kubernetes workloads to deploy applications:



it is time to explore how Kubernetes is able to manage persistent data.

# PERSISTING DATA FOR PODS

Deployments are stateless by design. Hence, any data stored inside the Pod’s container does not persist when the Pod dies.

If you were to update the content of the index.html file inside the container, and the Pod dies, that content will not be lost since a new Pod will replace the dead one.

Let us try that:

1. Scale the Pods down to 1 replica.

NAME READY STATUS RESTARTS AGE

nginx-deployment-56466d4948-58nqx 0/1 Terminating 0 45m

nginx-deployment-56466d4948-5z4c2 1/1 Terminating 0 45m

nginx-deployment-56466d4948-5zdbx 0/1 Terminating 0 62m

nginx-deployment-56466d4948-78j9c 1/1 Terminating 0 45m

nginx-deployment-56466d4948-gj4fd 1/1 Terminating 0 45m

nginx-deployment-56466d4948-gsrpz 0/1 Terminating 0 45m

nginx-deployment-56466d4948-kg9hp 1/1 Terminating 0 45m

nginx-deployment-56466d4948-qs29b 0/1 Terminating 0 45m

nginx-deployment-56466d4948-sfft6 0/1 Terminating 0 45m

nginx-deployment-56466d4948-sg4np 0/1 Terminating 0 45m

nginx-deployment-56466d4948-tg9j8 1/1 Running 0 62m

nginx-deployment-56466d4948-ttn5t 1/1 Terminating 0 62m

nginx-deployment-56466d4948-vfmjx 0/1 Terminating 0 45m

nginx-deployment-56466d4948-vlgbs 1/1 Terminating 0 45m

nginx-deployment-56466d4948-xctfh 0/1 Terminating 0 45m

NAME READY STATUS RESTARTS AGE

nginx-deployment-56466d4948-tg9j8 1/1 Running 0 64m

1. Exec into the running container (figure out the command yourself)
2. Install vim so that you can edit the file

apt-get update

apt-get install vim

1. Update the content of the file and add the code below /usr/share/nginx/html/index.html

<!DOCTYPE html>

<html>

<head>

<title>Welcome to DAREY.IO!</title>

<style>

body {

width: 35em;

margin: 0 auto;

font-family: Tahoma, Verdana, Arial, sans-serif;

}

</style>

</head>

<body>

<h1>Welcome to DAREY.IO!</h1>

<p>I love experiencing Kubernetes</p>

<p>Learning by doing is absolutely the best strategy at

<a href="https://darey.io/">www.darey.io</a>.<br/>

for skills acquisition

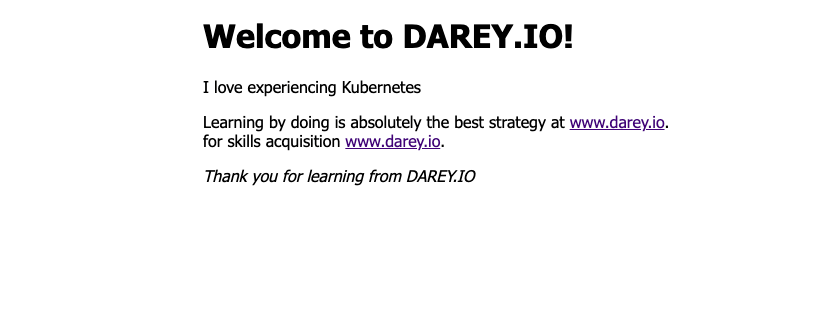
<a href="https://darey.io/">www.darey.io</a>.</p>

<p><em>Thank you for learning from DAREY.IO</em></p>

</body>

</html>

1. Check the browser – You should see this



1. Now, delete the only running Pod

kubectl delete po nginx-deployment-56466d4948-tg9j8

pod "nginx-deployment-56466d4948-tg9j8" deleted

1. Refresh the web page – You will see that the content you saved in the container is no longer there. That is because Pods do not store data when they are being recreated – that is why they are called ephemeral or stateless. (But not to worry, we will address this with persistent volumes in the next project)



Storage is a critical part of running containers, and Kubernetes offers some powerful primitives for managing it. **Dynamic volume provisioning**, a feature unique to Kubernetes, which allows storage volumes to be created on-demand. Without dynamic provisioning, DevOps engineers must manually make calls to the cloud or storage provider to create new storage volumes, and then create **PersistentVolume** objects to represent them in Kubernetes. The dynamic provisioning feature eliminates the need for DevOps to pre-provision storage. Instead, it automatically provisions storage when it is requested by users.

To make the data persist in case of a Pod’s failure, you will need to configure the Pod to use following objects:

* [**Persistent Volume**](https://kubernetes.io/docs/concepts/storage/persistent-volumes/) or pv – is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using [Storage Classes.](https://kubernetes.io/docs/concepts/storage/storage-classes/)
* [**Persistent Volume Claim**](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#expanding-persistent-volumes-claims) or pvc. Persistent Volume Claim is simply a request for storage, hence the "claim" in its name.

But where is it requesting this storage from?..

In the next project,

1. You will use Terraform to create a Kubernetes EKS cluster in AWS, and begin to use some powerful features such as **PV**, **PVCs**, **ConfigMaps**.
2. You will also be introduced to packaging Kubernetes manifests using [**Helm**](https://helm.sh/)
3. Experience Dynamic provisioning of volumes to make your Pods stateful, using Kubernetes Statefulset
4. Deploying applications into Kubernetes using Helm Charts
5. And many more awesome technologies

Keep it up!

