**Create Namespace**

kubectl create namespace development

[development is the name of namespace]

**Delete Namespace**

kubectl delete namespaces [Namespace name]

**Create Namespace through Yaml**

apiVersion: v1

kind: Namespace

metadata:

name: [namespace-name]

What is YAML file.

YAML Ain't Markup Language (YAML) is a serialization language that has steadily increased in popularity over the last few years. It's often used as a format for configuration files, but its object serialization abilities make it a viable replacement for languages like JSON.

Indentation and Whitespace

Whitespace is part of YAML's formatting. Unless otherwise indicated, newlines indicate the end of a field. You structure a YAML document with indentation. The indentation level can be one or more spaces.

YAML Deployment Example

**apiVersion**: apps/v1

**kind**: Deployment

**metadata**:

**name**: nginx-deployment

**spec**:

**selector**:

**matchLabels**:

**app**: nginx

**replicas**: 2 *# tells deployment to run 2 pods matching the template*

**template**:

**metadata**:

**labels**:

**app**: nginx

**spec**:

**containers**:

- **name**: nginx

**image**: nginx:1.14.2

**ports**:

- **containerPort**: 80

One way to create a Deployment using a .yaml file like the one above is to use the kubectl apply command in the kubectl command-line interface, passing the .yaml file as an argument. Here's an example:

kubectl apply -f <https://k8s.io/examples/application/deployment.yaml>

deployment.apps/nginx-deployment created

apiVersion - Which version of the Kubernetes API you're using to create this object

kind - What kind of object you want to create

metadata - Data that helps uniquely identify the object, including a name string, UID, and optional namespace

spec - What state you desire for the object

.spec.selector field defines how the created ReplicaSet finds which Pods to manage. In this case, you select a label that is defined in the Pod template (app: nginx). However, more sophisticated selection rules are possible, as long as the Pod template itself satisfies the rule.

/////////////////////////////////////////////////////////////////

YAML for Kubernetes

To get started with YAML for Kubernetes, we’ll review the basic structure types of YAML, which you can see in the table below:

When writing YAML files for Kubernetes, there are four required fields that must be present. APIVersion, Kind, Metadata, and Specifications.

**apiVersion**

This field refers to the API, which is being used to create the Kubernetes object being defined. Kubernetes provides various APIs that enable you to create different Kubernetes objects. For example, apiVersion: v1 contains many of the core objects.

apiVersion: v1 is usually considered the first stable release by Kubernetes. Another popular APIVersion is apps/v1, which adopts objects from v1 and provides crucial functionality such as deployments and ReplicaSets.

Hence, in our YAML file, defining the APIVersion could be:

apiVersion: v1

**kind**

kind allows you to specify which type of Kubernetes object you aim to define. The objects that you will specify in this field will be linked to the apiVersion that you specified before since it’s the APIVersion field that enables you to access the different types of objects and their specific definitions. Some of the types of objects that can be defined are pods, services, and DaemonSets.

So, to define a pod object, after specifying the apiVersion, we would specify the kind field as shown below:

apiVersion: v1

kind: pod

**metadata**

After specifying the type of object that’s being defined, the metadata field provides the unique properties for that specific object. This could include the name, uuid (**Universal Unique Identifier**) is a 128-bit value, and namespace fields. The values specified for these fields provide us with context for the object, and they can be referred to by other objects. So, this field allows us to specify the identifier properties of the object.

For example, if we are building a spring app, our pod could have the name value shown below:

apiVersion: v1

kind: Pod

metadata:

name: spring-pod

**spec**

The spec field allows us to define what’s expected from the object that we’re building. It consists of all the key-value pairs specific to defining the operation of the object. Just like the object itself**, the specifications of the object depend on the apiVersions** specified before. Hence, different APIVersions may include the same object, but the specifications of the object that can be defined will probably differ.

Example of building the pod object of our spring application, our spec field could resemble what you see below:

apiVersion: v1

kind: Pod

metadata:

name: spring-pod

containers:

– image: armo/springapp:example

spec:

name: spring-app

ports:

– containerPort: 80

protocol: TCP

In Example file above, we’re using the API v1 to create a Pod object, which we have named spring-pod. As per the specifications, the port we will use will be 80 and the image to be used is armo/springapp:example.

///////////////////////////////////

**Role of Deployment and Services**

////////////////////////////////////////////////////////////////

**Example ( Deployment Guest book php + mongo)**

//////////////////////////////////////////////////////////////

**Example:** Deploying PHP Guestbook application with MongoDB

This tutorial shows you how to build and deploy a simple (not production ready), multi-tier web application using Kubernetes and Docker. This example consists of the following components:

A single-instance MongoDB to store guestbook entries

Multiple web frontend instances

**Objectives**

Start up a Mongo database.

Start up the guestbook frontend.

Creating the Mongo Deployment

The manifest file, included below, specifies a Deployment controller that runs a single replica MongoDB Pod.

NOTE:

CPU time compared to if you asked for 1.0 CPU. For CPU resource units, the quantity expression 0.1 is equivalent to the expression 100m, which can be read as "one hundred millicpu". Some people say "one hundred millicores", and this is understood to mean the same thing.

Pay attention to the case of the suffixes. If you request 400m of memory, this is a request for 0.4 bytes. Someone who types that probably meant to ask for 400 mebibytes (400Mi) or 400 megabytes (400M).

**Deployment**

apiVersion: apps/v1

kind: Deployment

metadata:

name: mongo

labels:

app.kubernetes.io/name: mongo

app.kubernetes.io/component: backend

spec:

selector:

matchLabels:

app.kubernetes.io/name: mongo

app.kubernetes.io/component: backend

replicas: 1

template:

metadata:

labels:

app.kubernetes.io/name: mongo

app.kubernetes.io/component: backend

spec:

containers:

- name: mongo

image: mongo:4.2

args:

- --bind\_ip

- 0.0.0.0

resources:

requests:

cpu: 100m

memory: 100Mi

ports:

- containerPort: 27017

OR

kubectl apply -f <https://k8s.io/examples/application/guestbook/mongo-deployment.yaml>

kubectl get pods

**Creating the MongoDB Service**

The guestbook application needs to communicate to the MongoDB to write its data. You need to apply a Service to proxy the traffic to the MongoDB Pod. A Service defines a policy to access the Pods.

apiVersion: v1

kind: Service

metadata:

name: mongo

labels:

app.kubernetes.io/name: mongo

app.kubernetes.io/component: backend

spec:

ports:

- port: 27017

targetPort: 27017

selector:

app.kubernetes.io/name: mongo

app.kubernetes.io/component: backend

OR

kubectl apply -f <https://k8s.io/examples/application/guestbook/mongo-service.yaml>

kubectl get service

**Set up and Expose the Guestbook Frontend**

The guestbook application has a web frontend serving the HTTP requests written in PHP. It is configured to connect to the mongo Service to store Guestbook entries.

**Creating the Guestbook Frontend Deployment**

apiVersion: apps/v1

kind: Deployment

metadata:

name: frontend

labels:

app.kubernetes.io/name: guestbook

app.kubernetes.io/component: frontend

spec:

selector:

matchLabels:

app.kubernetes.io/name: guestbook

app.kubernetes.io/component: frontend

replicas: 3

template:

metadata:

labels:

app.kubernetes.io/name: guestbook

app.kubernetes.io/component: frontend

spec:

containers:

- name: guestbook

image: paulczar/gb-frontend:v5

# image: gcr.io/google-samples/gb-frontend:v4

resources:

requests:

cpu: 100m

memory: 100Mi

env:

- name: GET\_HOSTS\_FROM

value: dns

ports:

- containerPort: 80

OR

kubectl apply -f <https://k8s.io/examples/application/guestbook/frontend-deployment.yaml>

kubectl get pods -l app.kubernetes.io/name=guestbook -l app.kubernetes.io/component=frontend

**Creating the Frontend Service**

The mongo Services you applied is only accessible within the Kubernetes cluster because the default type for a Service is ClusterIP. ClusterIP provides a single IP address for the set of Pods the Service is pointing to. This IP address is accessible only within the cluster.

If you want guests to be able to access your guestbook, you must configure the frontend Service to be externally visible, so a client can request the Service from outside the Kubernetes cluster. However a Kubernetes user you can use kubectl port-forward to access the service even though it uses a ClusterIP.

apiVersion: v1

kind: Service

metadata:

name: frontend

labels:

app.kubernetes.io/name: guestbook

app.kubernetes.io/component: frontend

spec:

# if your cluster supports it, uncomment the following to automatically create

# an external load-balanced IP for the frontend service.

# type: LoadBalancer

ports:

- port: 80

selector:

app.kubernetes.io/name: guestbook

app.kubernetes.io/component: frontend

OR

kubectl apply -f <https://k8s.io/examples/application/guestbook/frontend-service.yaml>

kubectl get services

//////////////////////////////////////////////////////////////////////////

### Viewing the Frontend Service via kubectl port-forward

Run the following command to forward port 8080 on your local machine to port 80 on the service.

kubectl port-forward svc/frontend 8080:80

The response should be similar to this:

Forwarding from 127.0.0.1:8080 -> 80

Forwarding from [::1]:8080 -> 80

load the page [http://localhost:8080](http://localhost:8080/) in your browser to view your guestbook.

### Viewing the Frontend Service via LoadBalancer

If you deployed the frontend-service.yaml manifest with type: LoadBalancer you need to find the IP address to view your Guestbook.

Run the following command to get the IP address for the frontend Service.

kubectl get service frontend

The response should be similar to this:

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

frontend LoadBalancer 10.51.242.136 109.197.92.229 80:32372/TCP 1m

Copy the external IP address, and load the page in your browser to view your guestbook.

## Scale the Web Frontend

You can scale up or down as needed because your servers are defined as a Service that uses a Deployment controller.

Run the following command to scale up the number of frontend Pods:

kubectl scale deployment frontend --replicas=5

Query the list of Pods to verify the number of frontend Pods running:

kubectl get pods

The response should look similar to this:

NAME READY STATUS RESTARTS AGE

frontend-3823415956-70qj5 1/1 Running 0 5s

frontend-3823415956-dsvc5 1/1 Running 0 54m

frontend-3823415956-k22zn 1/1 Running 0 54m

frontend-3823415956-w9gbt 1/1 Running 0 54m

frontend-3823415956-x2pld 1/1 Running 0 5s

mongo-1068406935-3lswp 1/1 Running 0 56m

Run the following command to scale down the number of frontend Pods:

kubectl scale deployment frontend --replicas=2

Query the list of Pods to verify the number of frontend Pods running:

kubectl get pods

The response should look similar to this:

NAME READY STATUS RESTARTS AGE

frontend-3823415956-k22zn 1/1 Running 0 1h

frontend-3823415956-w9gbt 1/1 Running 0 1h

mongo-1068406935-3lswp 1/1 Running 0 1h