# A Comprehensive Documentation

## Index

* [Set up the Kubernetes Environment](#set-up-the-kubernetes-environment)
* [Database Connection](#database-connection)
* [Config Map and Secrets](#config-maps-and-sercrets)
* [Service Discovery](#service-discovery)

## Set up the Kubernetes Environment

Service mesh that enables users to describe network of microservices in discovery, load balancing, failure recovery, metrics and monitoring . Feature: - Traffic management - Security - Observability - Platform support(kubernetes, consul)

**Note: this guide primarily focus on seting up a single cluster via windows platform. However, the Kubernetes documentation offer a detailed insight in set up on a linux system.**

### 1. Prepare a virtualization environment.

Virtualization enviroment is necessary to allow kubernetes to run in [containerized environment](https://kubernetes.io/docs/tasks/tools/install-minikube/). First enable VT-x or AMD-v virtualization in computer's BIOS Usually, it is already configured by default. To [do that](https://docs.fedoraproject.org/en-US/Fedora/13/html/Virtualization_Guide/sect-Virtualization-Troubleshooting-Enabling_Intel_VT_and_AMD_V_virtualization_hardware_extensions_in_BIOS.html) 1. restart the computer 2. press F4 or F12 or any of the System menu key to enter the Bios menu 3. Look for "Chipset, advanced CPU Configuration" or "Northbridge" or othem similar items on different Bios 4. Enable Intel Virtualization Technoligy, or AMD-V depending on the processor make. 5. If options avaliable, enable Intel VTd, or AMD IOMMU which are used for PCI passthrough (allows hypervisor) 6. remember to save before exiting the Bios menu

### 2. Install a Hypervisor

Windnows already have a hypervisor Hyper-V, but install the type of hyperviSor you prefer based on your operation system. - macOS: VMware Fusion, HyperKit; linux: KVM; Windows: hyper-V, all: virtualBox follow the installation wizard to finish the set up

### 3. Prepare a Docker environment

Register and [install docker](https://www.docker.com/). Docker windows for windows enterprise users, docker tool box for other windows users Or install docker according to your local OS.

### 3.1 Enable Kubernetes on Docker environment

on Docker for window and Docker for Mac, Kubernetes could be installed along with docker; If using Docker Toolbox, or any other version of Docker that does not come with kubernetes integration, skip this step. Upon finishing the setup, right click on the icon on the task menu, then navigate to settings->kubernetes

Choose to enable Kubernetes According to personal preference select or deselect Kubernetes as "Default orchestrator for docker stack commands"

**Note: This installation configures 'Docker for %{YOUR\_OS\_HERE}' as default context for Kubernetes, if want to run kubernetes in minikube context, right click on the Docker icon -> kubenetes -> d**

### 4. Minikube:

Minikube is recommended to work with a kubernetes app on a **single cluster single node on local machine in virtualized environment**. Get Started on a single cluster on local machine, go to https://github.com/kubernetes/minikube/releases, and download and run the installer.

### 5. install Kubernetes

If 3.1 is done, skip this step. To install kubernetes, install kubectl by through a command line install such as 'choco install kubernetes-cli', then run 'kubectl cluster-info' to see if kubernetes is up and running.

### 4. deploy to Cloud service

Kubernetes allow deployment to cloud platforms. [kubeadm](https://kubernetes.io/docs/setup/independent/install-kubeadm/) is a tool to bootstrap a minimum kubernetes cluste

# Database Connection

Databse on Kubernetes can be set up in mostly 2 ways: 1. Set up the database as a cluster on the kubernetes container. 2. Set up the databse somewhere else.

## Set up Databse on an External VM with Istio Mesh Expansion

This is the instance where the database is hosted at an external location. Most likely, this option is used to connect new microservice apps to use service Entry;

1. Have a database already set up at desired location by running the .sql set up file.
2. Obtain the host and ports of the databse. If it is on a linux machine, you can run commands:

$export MYSQL\_DB\_HOST=<your MySQL database host>  
 $export MYSQL\_DB\_PORT=<your MySQL database port>

Setup a table if you will, run a table initializer like this:

CREATE DATABASE test;  
USE test;  
  
CREATE TABLE `ratings` (  
 `ReviewID` INT NOT NULL,  
 `Rating` INT,  
 PRIMARY KEY (`ReviewID`)  
);  
INSERT INTO ratings (ReviewID, Rating) VALUES (1, 5);  
INSERT INTO ratings (ReviewID, Rating) VALUES (2, 4);

The SQL file above

1. For the sake of development, you could grant access to a specific user or all users using:

GRANT ALL PRIVILEGES ON \*.\* TO 'root'@'localhost' IDENTIFIED BY 'password' WITH GRANT OPTION;  
quit;

1. In the deployment yaml of the application that reference the database table, enter the following configuration to supply connection details.

apiVersion: extensions/v1beta1  
kind: Deployment  
metadata:  
 name: friend-service  
spec:  
 replicas: 1  
 template:  
 metadata:  
 labels:  
 app: friend-service  
 version: v2-mysql-vm  
 spec:  
 containers:  
 - name: ratings  
 image: friend-service:v0  
 imagePullPolicy: IfNotPresent  
 env:  
 # This assumes you registered your mysql vm as  
 # istioctl register -n vm mysqldb 1.2.3.4 3306  
 - name: DB\_TYPE  
 value: "mysql"  
 - name: MYSQL\_DB\_HOST  
 value: mysqldb.vm.svc.cluster.local  
 - name: MYSQL\_DB\_PORT  
 value: "3306"  
 - name: MYSQL\_DB\_USER  
 value: root  
 - name: MYSQL\_DB\_PASSWORD  
 value: password  
 ports:  
 - containerPort: 9080

The 'env' section sets the environment varible for the friend-service, and is also where the necessary parametes are supplied.

1. Register istio with the location of the VM on which the database is hosted:

$istioctl register -n vm mysqldb <ip-address-of-vm> 3306`

where

-n <name-of-VM> <ip-address-of-vm> <port>

## Set up Databse on an External VM with external TCP connection

Replace step 4 and onwards with the following setup:

1. Write a Service Entry that would detail the connection host, path, and location in the deployment.yaml file

apiVersion: networking.istio.io/v1alpha3  
kind: ServiceEntry  
metadata:  
 name: mysql-external  
spec:  
 hosts:  
 - $MYSQL\_DB\_HOST  
 addresses:  
 - $MYSQL\_DB\_IP/32  
 ports:  
 - name: tcp  
 number: $MYSQL\_DB\_PORT  
 protocol: tcp  
 location: MESH\_EXTERNAL

### Comparison of Mesh Expansion and Service Entry

Due to the containerized nature of istio environment, it is not able to discover services outside of the mesh. Hence it is necessary to a service entry to allow access to the outside components. This is different than in the previous example of 'Set up Databse on an External VM with Istio Mesh Expansion', where the command 'istioctl register ' performs a mesh expansion to include the external service which elevate the ability to enjoy any feature internal services have.

In the case of Istio Mesh Expansion, the Istio component(Envoy proxy, node-agent, istio-agent) must be added to the external databse, and the istio's control plane (Pilot, Mixer, Citadel) must be able to access it.

Because mesh expansion includes the external database into the mesh, the traffic into the database are still regarded as internal traffic. Consequently, this effectively avoided exiting bottle neck and service entry configuration

However, there is a warning given by Istio that:

**Warning Mesh expansion is broken in the initial release of Istio 1.0. We expect to have a patch available that fixes this problem very soon. We apologize for the inconvenience.**

So far the most convinent way is to utilize external TCP through service entry to connect to an external DB.

## Spawn database as a Pod in kubernetes Cluster

A databse can be spawned on a kubernetes pod similar to the way a service is done. It is generally not recommended though, to store a stateful database on a stateless kubernetes cluster. This is because a database is stateful, where it would require the state of the media to complete its task. A stateless container does not offer offers the ability to be quickly brought online or off, which is one that the databse do not need. On the other hand, the container would restrict the throughoutput to decrease the speed of database processing. It is best use to store a small amount of data for immediate use for transient purpose. For example, developing, testing, temperary data storing.

## SQL VS NoSQL databse

SQL is relational and NoSQL is non-relational. SQL stores data in tables, so to store and access data of an object, multiple sql tables may be consulted. On the other hand NoSQL store data into an object as it is. Let's say we want to store an object called friend that has their name, address, and gender, as well as their friends:

friend: {  
 name: "json",  
 gender: "binary",  
 address: "0x677882"  
  
 friends: [  
 "xml",  
 "soap",  
 "yaml"  
 ]  
  
}

In NoSQL it will be store as it is, in SQL, it will most likely be store into 2 tables:

'friend' table

|  |  |  |  |
| --- | --- | --- | --- |
| id | name | gender | address |
| 1 | json | binary | 0x677882 |
| 2 | soap | ... | ... |
| 3 | ... | ... | ... |

'friends' table

|  |  |
| --- | --- |
| id | friends |
| 1 | 2,3,4 |
| 2 | 1,5,3 |
| 3 | 7,9,4 |

It is recommended, to store database connection detail in config map, and connection authentication in

# Config Maps and Sercrets

Both Configure maps and secrets provide environment variables to be used in the cluster that one would otherwise not want to have directly hard coded in each container's deployment file. This could be for a number of reasons; 1. The Pod Creator does not know the value of the environmental variable but will get it from some one. 2. The variable will be shared across multiple containers. 3. The value will want to be hidden from the user (use secrets)

## Store Reference to Microservices in the Config Map

The Following Config Map stores the reference to the service path:

apiVersion: v1  
kind: ConfigMap  
metadata:  
 name: path-config  
 namespace: default  
data:  
 ACCOUNT\_SERVICE: "http://account"  
 FRIEND\_SERVICE: "http://friend-service"  
 ENEMY\_SERVICE: "http://enemy-service"

Then Deployment imports the environment variable into the pod:

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 ...  
spec:  
 ...  
 template:  
 ...  
 spec:  
 containers:  
 - name: bff  
 image: bff:v0  
 imagePullPolicy: IfNotPresent  
 ...  
 envFrom:  
 - configMapRef:  
 name: path-config

## Store Environmental Variable referenced Across Services and Pods

The following file stores a number of gloabal variabkles for the game

apiVersion: v1  
data:  
 game.properties: |  
 enemies=aliens  
 lives=3  
 enemies.cheat=true  
 enemies.cheat.level=noGoodRotten  
 secret.code.passphrase=UUDDLRLRBABAS  
 secret.code.allowed=true  
 secret.code.lives=30  
 ui.properties: |  
 color.good=purple  
 color.bad=yellow  
 allow.textmode=true  
 how.nice.to.look=fairlyNice  
kind: ConfigMap  
metadata:  
 creationTimestamp: 2016-02-18T18:52:05Z  
 name: game-config  
 namespace: default  
 resourceVersion: "516"  
 selfLink: /api/v1/namespaces/default/configmaps/game-config  
 uid: b4952dc3-d670-11e5-8cd0-68f728db1985

## Store Environmental Variables to manipulate Pod set up

The following config map is used to set up the reddis cache

apiVersion: v1  
data:  
 redis-config: |  
 maxmemory 2mb  
 maxmemory-policy allkeys-lru  
kind: ConfigMap  
metadata:  
 creationTimestamp: 2016-03-30T18:14:41Z  
 name: example-redis-config  
 namespace: default  
 resourceVersion: "24686"  
 selfLink: /api/v1/namespaces/default/configmaps/example-redis-config  
 uid: 460a2b6e-f6a3-11e5-8ae5-42010af00002

Then The Raddis Deployment refer to the 'example-radis-config' to intialize the pod. Any other database, persistent volumne is set up pretty much the same way.

apiVersion: v1  
kind: Pod  
metadata:  
 name: redis  
spec:  
 containers:  
 - name: redis  
 image: kubernetes/redis:v1  
 env:  
 - name: MASTER  
 value: "true"  
 ports:  
 - containerPort: 6379  
 resources:  
 limits:  
 cpu: "0.1"  
 volumeMounts:  
 - mountPath: /redis-master-data  
 name: data  
 - mountPath: /redis-master  
 name: config  
 volumes:  
 - name: data  
 emptyDir: {}  
 - name: config  
 configMap:  
 name: example-redis-config  
 items:  
 - key: redis-config  
 path: redis.conf

## Container profile vs Local Profile.

Spring allows you to set up specific profiles. The variables in the application.properties or application.yml falls into the default profile. To enable container profile, set container entry point at

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","-Dspring.profiles.active=container", "/app.jar"]

Where the '-Dspring.profiles.active=container' will activate the container profile. Now you can have two set of variables for uncontainerized and containerized environment.

spring:  
 jpa:  
 generate-ddl: true  
 hibernate:  
 ddl-auto: none  
 show-sql: true  
 application:  
 name: friend-service  
 datasource:  
 url: jdbc:mysql://den1.mysql6.gear.host/coolfriends  
 username: coolfriends  
 password: Ab77F-88M\_9l  
 data:  
 rest:  
 base-path: api  
---  
spring:  
 profiles: container  
 datasource:  
 url: jdbc:mysql://den1.mysql6.gear.host/coolfriends  
 host: den1.mysql6.gear.host/32  
 username: ${DATABASE\_USER}  
 password: ${DATABASE\_PASSWORD}  
server:  
 port: 8081  
  
The first part of spring is the uncontainerized set up, and the second part is the containerized profile.

## Secres

A secret is an object that stores sensitive data in an encoded fashion. Secret stores base64 encoded data in a map:

apiVersion: v1  
kind: Secret  
metadata:  
 name: mysecret  
type: Opaque  
data:  
 username: YWRtaW4=  
 password: MWYyZDFlMmU2N2Rm

The String can be encoded:

$ echo -n 'admin' | base64  
YWRtaW4=  
$ echo -n '1f2d1e2e67df' | base64  
MWYyZDFlMmU2N2Rm

and decoded:

$ echo 'MWYyZDFlMmU2N2Rm' | base64 --decode  
1f2d1e2e67df

Then import it as environment Variable:

- name: SECRET\_PASSWORD  
 valueFrom:  
 secretKeyRef:  
 name: mysecret  
 key: password

# Service Discovery

## Kube-DNS, Ingress, Envoy

By default, Kubernetes, and the istio on top of it uses Kube-DNS to assign a name to the Microservice Service. The name is named in the selector section of service deployment. Then when traffic comes through the Ingress Gateway, Envy performs a service discovery to route user to the desired location. In the mean time, performing load balancing and security.

Here are the two major appraoches through default method:

### Container Level Port Forwarding

This is the most commonly used method found online. The Approach was to rely on Kube-DNS to poiint to the microservice's default ':80' port. Then in service route it to the application port.

kind: Service  
apiVersion: v1  
metadata:  
 name: my-service  
spec:  
 selector:  
 app: MyApp  
 ports:  
 - name: http  
 protocol: TCP  
 port: 80  
 targetPort: 9376  
 externalIPs:  
 - 80.11.12.10

Here the deployment container accept connection on default 80 port.Then forward the connection to port '9376' on the application level.

This method works the best when each microservice correspond to one port. The deployer of this microservice knows which port they shall forward to on the application level. So that when developer calls a service, they do not need to be concern about the port to call.

However, this method force every call on container level to 80.

### Port Naming

Aside from container level port forwarding, one can name the port at of "http" and run a query for DNS SRV record by "\_http.\_tcp.my-service.my-ns" to discover the port "http".

### DNS SRV Record

## CoreDNS

Core DNS is a replacement to the kube-DNS, it offers "chained plugin", which means, it supports mutiple plugins, which also means, it opens up the option to program customizable plugin. CoreDNS could be used to give a higer degree of freedom when we need to handle our own DNS service discovery.

## Test on DNS Records:

run this:

$ kubectl run -i -t --image=infoblox/dnstools:k8sblog --restart=Never dnstest  
  
Waiting for pod default/dnstest to be running, status is Pending, pod ready: false  
If you don't see a command prompt, try pressing enter.  
  
# curl nginx.default.svc.cluster.local

# Autoscale

Kubernetes supports Auto-scaling out of the box. However, auto-scale depends on metrics server to determine when to trigger the scaling operation.

To get set an existing deployment to autoscale, run

kubectl autoscale <pod\_name> -

Autoscale failed because of unable to detect the CPU useage: "<unknown>/50%"

This is because Kubernetes does not have a metrics server or unable to utilize its metrics server. To install a kubernetes metrics server, run: yaml and error occurs for "CPU: <unknown>/50%" kubectl apply -f https://raw.githubusercontent.com/kubernetes/kops/master/addons/metrics-server/v1.8.x.yaml