KUBERNETES

JOHN BRYCE
Leading in IT Education
a matrix company

About me

https://www.opsguru.io/

lev@opsguru.io

Lev Epshtein

Technology enthusiast with 10 years of industry experience in DevOps and IT. Industry experience with back-end architecture.

Solutions architect with experience in big scale systems hosted on AWS/GCP, end-to-end DevOps automation process.

DevOps, and Big Data instructor at John Bryce.

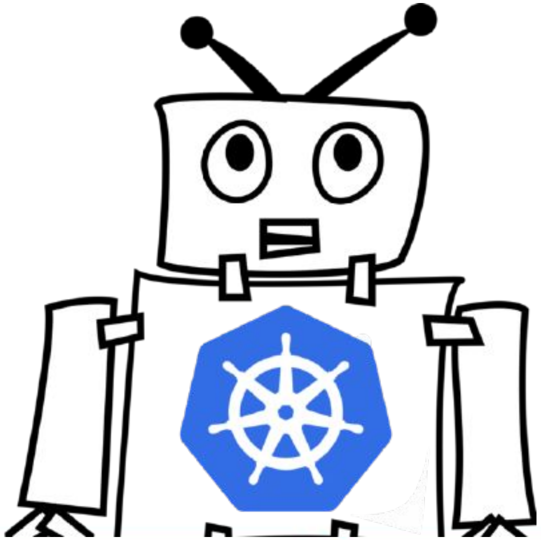
Partner & Senior Engineering Consultant at Opsguru.

What is Kubernetes?

Kubernetes is a portable, extensible open-source platform for managing containerized workloads and services, that facilitates both **declarative configuration** and **automation**. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

Why K8S?

Developers are ***lazy*** and somewhere in the mid-late 80s they started abbreviating the words based on their first letter, last letter, and number of letters in between. This is why you'll sometimes see i18n for internationalization and l10n for localization. There are also new numeronyms such as Andreessen Horowitz (a16z) and of course our favorite k**ubernete**s (**k8s**).



Why do we need Kubernetes?

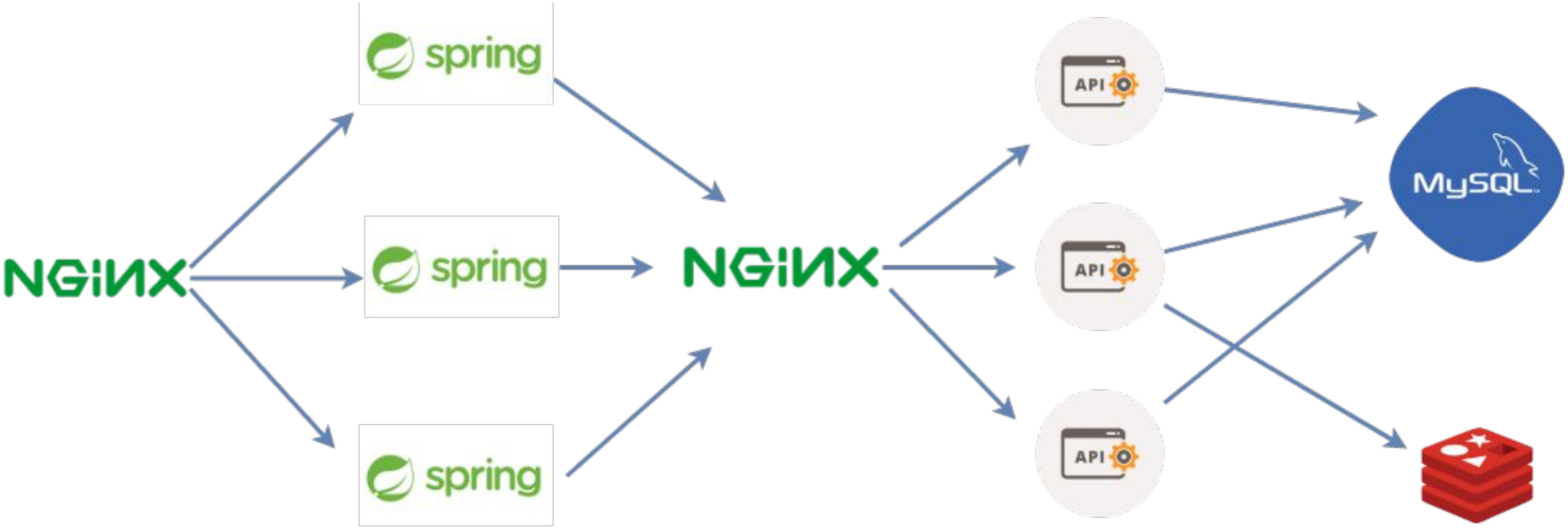
- a container platform
- a microservices platform
- a portable cloud platform and a lot more.
- Kubernetes provides a container-centric management environment. It orchestrates computing, networking, and storage infrastructure on behalf of user workloads. This provides much of the simplicity of Platform as a Service (PaaS) with the flexibility of Infrastructure as a Service (IaaS), and enables portability across infrastructure providers.
- K8S Allows developers / system operators to cut to the cord and truly run a container-centric dev / microservice environment

MICROSERVICES

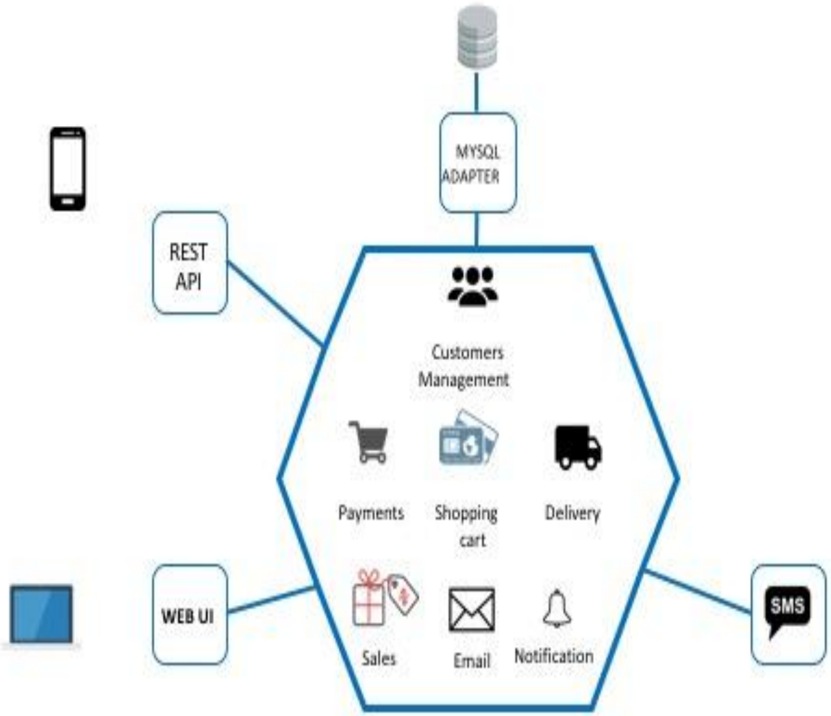
MICROSERVICES

Small decoupled services / API Layer

No shared libraries / common code

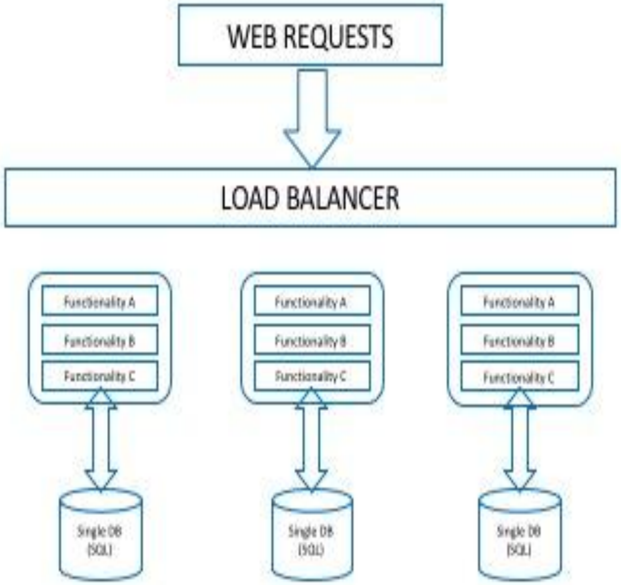


MONOLITHIC ARCHITECTURE



SCALING MONOLITHIC APPLICATION

The whole application scales under LB and not a single component



MICROSERVICES

CONS MONOLITHIC APPLICATION

- A nightmare to manage in large scale due to the fact that developers keeps on adding new features and changes which makes it complex and difficult to fully understand what does what
- Everything is written and coded in the same programing language
- Each deploy -> full deploy to the whole system
- Reliability and Bug hunting (Full regression anyone?)
- Change / Upgrade framework - Easier to land on the moon (again)

Kubernetes is not completely PaaS

Kubernetes is **not a traditional, all-inclusive PaaS** (Platform as a Service) system (Incounterary to OpenShift).

Kubernetes operates at the container level rather than at the hardware level, it provides some generally applicable features **common** to PaaS offerings, such as **deployment**, **scaling**, **load balancing**, **logging**, and **monitoring**.

Kubernetes is not completely PaaS

- Does not deploy source code and does not build your application - **CI/CD Does**
- Does not provide application-level services, such as middleware (e.g., message buses), data-processing frameworks (for example, Spark), databases (e.g., mysql), caches, nor cluster storage systems (e.g., Ceph) as built-in services
- Does not dictate logging, monitoring, or alerting solutions.
- Does not provide nor mandate a configuration language/system (e.g., jsonnet). It provides a declarative API that may be targeted by arbitrary forms of declarative specifications.
- Does not provide nor adopt any comprehensive machine configuration, maintenance, management, or self-healing systems.

K8S: Concepts

To work with Kubernetes, we use Kubernetes API objects to describe our cluster's desired state:

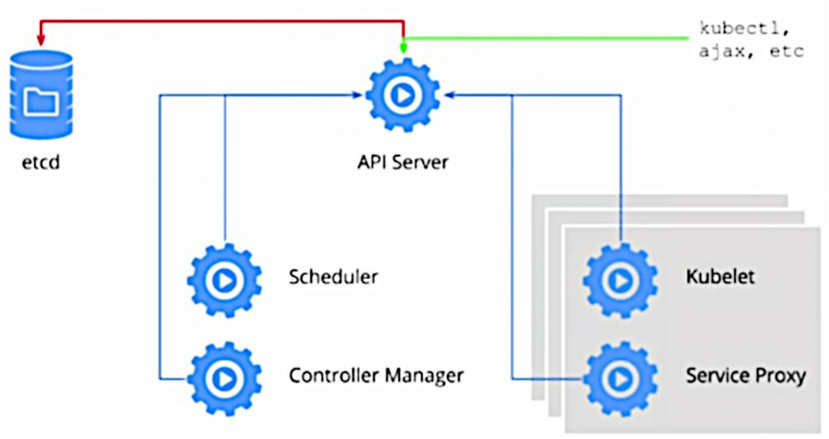
- what applications or other workloads we want to run
- what container images they use, the number of replicas, what network and disk resources we want to make available.
- setting our desired state by creating objects using the Kubernetes API (typically via the command-line interface - kubectl)

Once we've set our desired state, Kubernetes Control Plane works to make the **cluster's current state match the desired state**

K8S: Concepts

Kubernetes performs a variety of tasks automatically–such as starting or restarting containers, scaling the number of replicas of a given application, and more. The **Kubernetes Control Plane** consists of a collection of processes running on your cluster:

- **Kubernetes Master**:
  - etcd
  - kube-apiserver
  - kube-controller-manager
  - kube-scheduler
- **Non-master**:
  - kubelet
  - kube-proxy



K8S: Concepts

etcd - highly-available key value store used as Kubernetes' backing store for all cluster data

kube-apiserver - Validates and configures data for the api objects which include pods, services, replicationcontrollers, and others. Exposes the Kubernetes API. It is the front-end for the Kubernetes control plane. It is designed to scale horizontally – that is, it scales by deploying more instances.

kube-scheduler - its job is to take pods that aren't bound to a node, and assign them one along with hardware/software/policy constraints

K8S: Concepts

kube-controller-manager - is a an application control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state:

- **Node Controller** - Responsible for noticing and responding when nodes go down
- **Replication controller** - Responsible for maintaining the correct number of pods for every replication controller object in the system
- **Endpoints Controller** - Populates the Endpoints object (that is, joins Services & Pods)
- **Service account & Token Controllers** - Create default accounts and API access tokens for new namespaces

K8S: Concepts

Cloud-controller-manager - runs controllers that interact with the underlying cloud providers. cloud-controller-manager allows cloud vendors code and the Kubernetes core to evolve independent of each other and develops functionality (by the cloud providers) that will be linked to K8S cloud-controller-manger. The following controllers have cloud provider dependencies

- **Node Controller**: For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding
- **Route Controller**: For setting up routes in the underlying cloud infrastructure
- **Service Controller**: For creating, updating and deleting cloud provider load balancers
- **Volume Controller**: For creating, attaching, and mounting volumes, and interacting with the cloud provider to orchestrate volumes

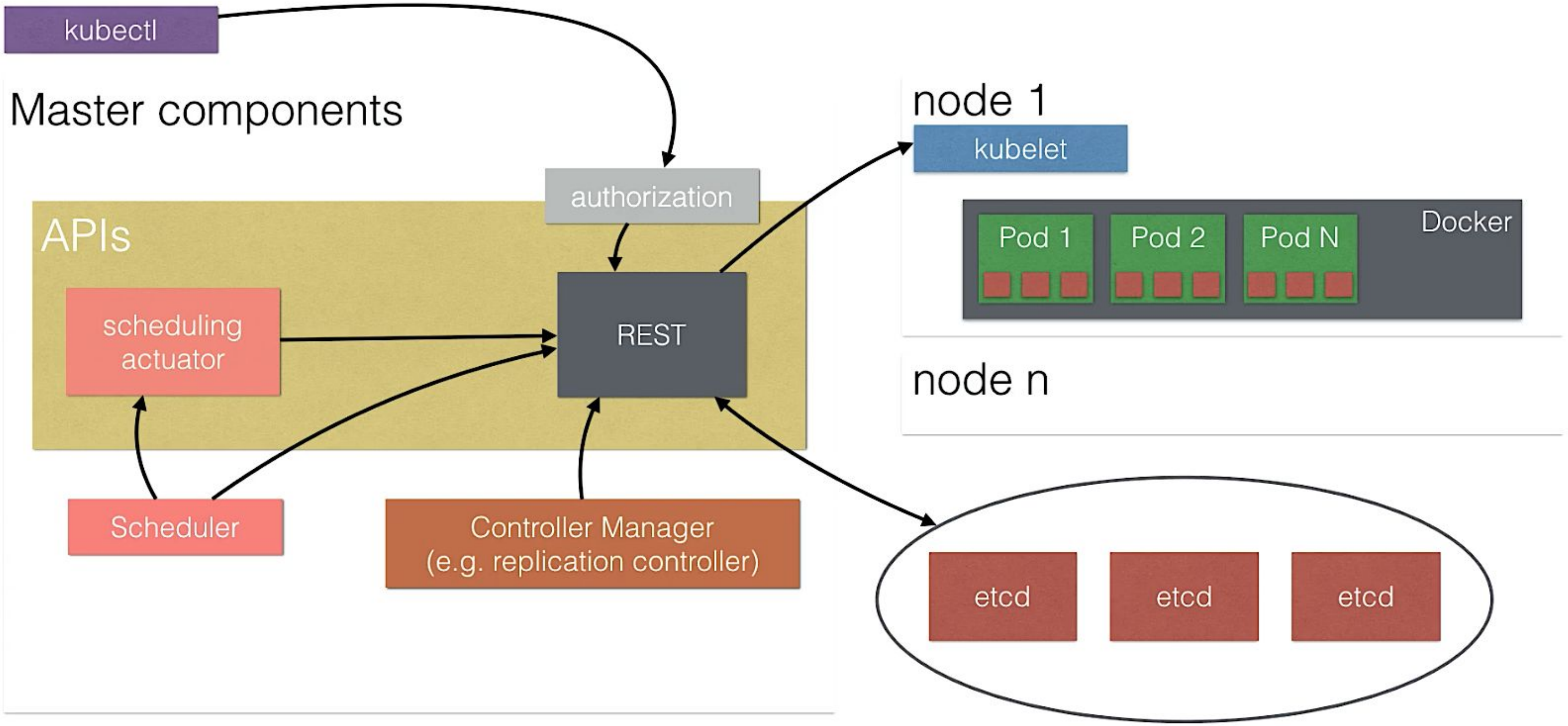
K8S: Concepts

On each non-master node:

kubelet - which communicates with the Kubernetes Master.

kube-proxy - A network proxy which reflects Kubernetes networking services on each node.

K8S: Concepts



K8S: Objects

Kubernetes contains a number of abstractions that represent the state of your system: deployed containerized applications and workloads, their associated network and disk resources and other :

:The basic Kubernetes objects include

- **Pod**
- **Service**
- **Volume**
- **Namespase**

:The higher-level abstractions called Controllers

- **ReplicaSet**
- **Deployment**
- **StatuefulSet**
- **DaemonSet**
- **Job**

K8S: Objects

For this current course we have already created the Kubernetes environment on CentOS for you, please ask mentor to provide the credentials.

⌨ *by this sign we would mark the practice part*

*Get to the course Kubernetes **master** environment using SSH CLI (command line :(interface*

Type the following command to get the kubectl version and cluster configuration:

```
kubectl --version<
kubectl cluster-info<
```

The output should be like this:

```
Kubernetes v1.9.2+coreos.0<
Kubernetes master is running at http://localhost:8080 KubeDNS is running at<
http://localhost:8080/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
```

K8S: Objects

Type the following command to get cluster nodes configuration:

kubectl get nodes*<*

```
NAME                                           STATUS   ROLES    AGE   VERSION
ip-172-24-2-10.eu-central-1.compute.internal   Ready    master   23h   v1.9.2+coreos.0
ip-172-24-3-10.eu-central-1.compute.internal   Ready    node     23h   v1.9.2+coreos.0
ip-172-24-3-11.eu-central-1.compute.internal   Ready    node     23h   v1.9.2+coreos.0
```

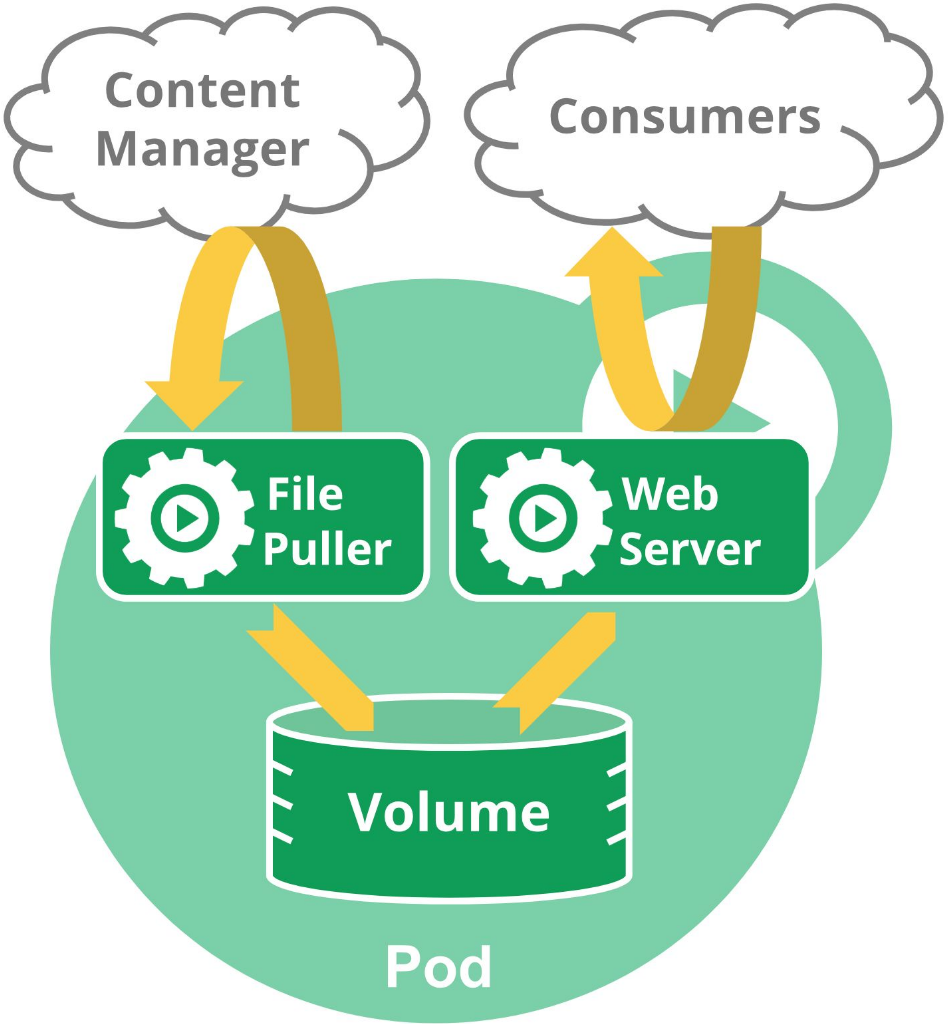
K8S: Objects

Pod is the basic building block of Kubernetes–the smallest and simplest unit in the Kubernetes object model that you create or deploy. A Pod represents a running process on your cluster.

Pods in a Kubernetes cluster can be used in two main ways:

- **Pods that run a single container.** The "one-container-per-Pod" model is the most common Kubernetes use case
- **Pods that run multiple containers that need to work together**. A Pod might encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources.

K8S: Objects


Content Manager
Consumers
File Puller
Web Server
Volume
Pod

K8S: Objects

To get the pods list, please run the following on the master node:

kubectl get pods<

We have already installed the test application there- Nexus repository manager, so the output should be like this:

NAME	READY	STATUS	RESTARTS	AGE
edgy-grizzly-sonatype-nexus-6d6bf984f-stbxb	1/1	Running	0	25m

**-please note, that the name ID part should differ per item:*

edgy-grizzly-sonatype-nexus-6d6bf984f-stbxb

K8S: Objects

*To see the **details** about the object, please run:*

```
kubectl describe pod edgy-grizzly-sonatype-nexus-6d6bf984f-stbxb<
```

*To see the object’s **logs**, please run:*

```
kubectl logs edgy-grizzly-sonatype-nexus-6d6bf984f-stbxb<
```

*To **delete*** the pod, please run:*

```
kubectl delete pod edgy-grizzly-sonatype-nexus-6d6bf984f-stbxb<
```

- no worries, the pod would be deleted, albeit the **Replica set would start the other one with the new ID. Check this by **kubectl get pods***

K8S: Objects

On-disk files in a Container are ephemeral, which presents some problems for non-trivial applications when running in Containers.

- when a Container crashes, kubelet will restart it, but the files will be lost - the Container starts with a clean state.
- when running Containers together in a Pod it is often necessary to share files between those Containers.

The Kubernetes **Volume** abstraction solves both of these problems.

K8S: Objects

Kubernetes supports several types of Volumes:

- awsElasticBlockStore
- azureDisk
- azureFile
- cephfs
- configMap
- csi
- downwardAPI
- emptyDir
- fc (fibre channel)
- flocker
- gcePersistentDisk
- gitRepo (deprecated)
- glusterfs
- hostPath
- iscsi
- local
- nfs
- persistentVolumeClaim
- projected
- portworxVolume
- quobyte
- rbd
- scaleIO
- secret
- storageos
- vsphereVolume

K8S: Objects

Let’s elaborate some of them:

ConfigMap volume- resource provides a way to inject configuration data into Pods. The data stored in a ConfigMap object can be referenced in a volume of type configMap and then consumed by containerized applications running in a Pod.

Basically, the ConfigMap mechanism stores the alias mapping for the parameters to keep containerized applications portable.

K8S: Objects

A **Secret volume** is used to pass sensitive information, such as passwords, to Pods. You can store secrets in the Kubernetes API and mount them as files for use by Pods without coupling to Kubernetes directly. secret volumes are backed by tmpfs (a RAM-backed filesystem) so they are never written to non-volatile storage.

K8S: Objects

A **PersistentVolumeClaim volume** is used to mount a PersistentVolume into a Pod. PersistentVolumes are a way for users to “claim” durable storage (such as a GCE PersistentDisk or an iSCSI volume) without knowing the details of the particular cloud environment. Basically it’s like a shortcut to the external provider.

K8S: Objects

An **NFS volume** allows an existing NFS (Network File System) share to be mounted into your Pod. The contents of an nfs volume are preserved and the volume is merely unmounted when the pod is removed. This means that an NFS volume can be pre-populated with data, and that data can be "handed off" between Pods. NFS can be mounted by multiple writers simultaneously.

K8S: Objects

To get the pod’s volume information, please run:

kubectl describe pod edgy-grizzly-sonatype-nexus-6d6bf984f-stbxb<

And see the “Volumes” section:

```
…
:Volumes
:nexus-data-volume
(Type:      PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the same namespace
ClaimName:  edgy-grizzly-sonatype-nexus
ReadOnly:   false
:default-token-7gr5x
(Type:       Secret (a volume populated by a Secret
SecretName:  default-token-7gr5x
Optional:    false
…
```

K8S: Objects

Let’s elaborate the volumes, please run:

kubectl get pvc*<*

For the PersistentVolumeClaim#

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	AGE
edgy-grizzly-sonatype-nexus	Bound	nexus	10Gi	RWO		10h

kubectl get secrets*<*

NAME	TYPE	DATA	AGE
default-token-7gr5x	kubernetes.io/service-account-token	2	94d

*Run ‘**describe’** on the objects to see the details*

K8S: Objects

PVs are resources in the cluster. **PVCs** are requests for those resources and also act as claim checks to the resource. The interaction between PVs and PVCs follows this lifecycle. There are two ways PVs may be provisioned:

Static

A cluster administrator creates a number of PVs. They carry the details of the real storage which is available for use by cluster users. They exist in .the Kubernetes API and are available for consumption

Dynamic

When none of the static PVs the administrator created matches a user's PersistentVolumeClaim, the cluster may try to dynamically .provision a volume specially for the PVC

K8S: Objects

To get the original PV our PVC has been mapped from, please run:

kubectl get pv<

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM	STORAGECLASS	REASON	AGE
nexus	10Gi	RWO	Recycle	RWO	default/edgy-grizzly-sonatype-nexus			10h

kubectl describe pv nexus<

```
:Source
(Type:      NFS (an NFS mount that lasts the lifetime of a pod
Server:    172.24.2.10
Path:      /nfs/nexus
ReadOnly:  false
```

*Run the : **ls -ltr /nfs/nexus** to see the directory on the hosed Centos7 machine*

K8S: Objects

Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called **namespaces**.

Namespaces are intended for use in environments with many users spread across multiple teams, or projects. For clusters with a few to tens of users, you should not need to create or think about namespaces at all. Start using namespaces when you need the features they provide.

Namespaces provide a scope for names. Names of resources need to be **unique** within a namespace, but not across namespaces.

K8S: Objects

You can list the current namespaces in a cluster using:

```
kubectl get namespaces<
```

```
NAME        STATUS   AGE
default      Active   94d
kube-public   Active    94d
kube-system   Active    94d
```

- **default** The default namespace for objects with no other namespace
- **kube-system** The namespace for objects created by the Kubernetes system
- **kube-public** This namespace is created automatically and is readable by all users (including those not authenticated). This namespace is mostly reserved for cluster usage, in case that some resources should be visible and readable publicly throughout the whole cluster. The public aspect of this namespace is only a convention, not a requirement.

K8S: Objects

To temporarily set the namespace for a request, use the **--namespace** flag:

```
>kubectl --namespace=default get pods
```

K8S: Objects

A Kubernetes **Service** is an abstraction which defines a logical set of Pods and a policy by which to access them - sometimes called a **micro-service**. The set of Pods targeted by a Service is (usually) determined by a Label Selector.

The issue a service solves:

While each Pod gets its own IP address, even those IP addresses cannot be relied upon to be stable over time. This leads to a problem when some set of Pods provides functionality to other Pods inside the Kubernetes cluster.

K8S: Objects

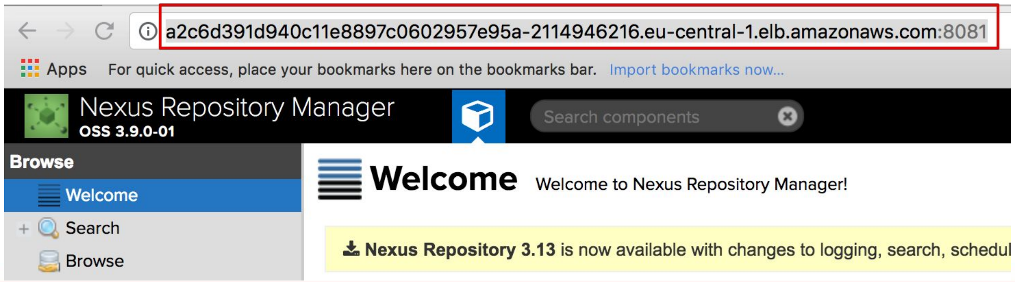
You can list the current services:

kubectl get svc<

kubectl describe svc edgy-grizzly-sonatype-nexus<

```
…
Selector: app=sonatype-nexus,release=edgy-grizzly
LoadBalancer Ingress:
a2c6d391d940c11e8897c0602957e95a-2114946216.eu-central-1.elb.amazonaws.com
Port: nexus 8081/TCP
TargetPort: 8081/TCP
…
```

The current service maps the AWS load balancer to the sonatype-nexus pod via 8081 port, you could ensure that by:* ***http://a2c6d391d940c11e8897c0602957e95a-2114946216.eu-central-1.elb.amazonaws.com:8081/***



a2c6d391d940c11e8897c0602957e95a-2114946216.eu-central-1.elb.amazonaws.com:8081

For quick access, place your bookmarks here on the bookmarks bar. Import bookmarks now...

Welcome to Nexus Repository Manager!

Nexus Repository 3.13 is now available with changes to logging, search, schedul

K8S: Objects

Kubernetes ServiceTypes:

- **ClusterIP**: Exposes the service on a cluster-internal IP. Choosing this value makes the service only reachable from within the cluster. This is the default ServiceType.
- **NodePort**: Exposes the service on each Node’s IP at a static port (the NodePort). A ClusterIP service, to which the NodePort service will route, is automatically created. You’ll be able to contact the NodePort service, from outside the cluster, by requesting <NodeIP>:<NodePort>.
- **LoadBalancer**: Exposes the service externally using a cloud provider’s load balancer. NodePort and ClusterIP services, to which the external load balancer will route, are automatically created.
- **ExternalName**: Maps the service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record with its value. No proxying of any kind is set up. This requires version 1.7 or higher of kube-dns.

K8S: Objects

A **ReplicaSet** ensures that a specified number of pod replicas are running at any given time. Kubernetes pods are mortal. They are born and when they die, they are not resurrected. ReplicaSets in particular create and destroy Pods dynamically (e.g. when scaling up or down).

K8S: Objects

To get the ReplicaSet list, please run:

kubectl get replicasets*<*

NAME	DESIRED	CURRENT	READY	AGE
edgy-grizzly-sonatype-nexus-6d6bf984f	1	1	1	2h

Please note that the replicaset name is mentioned in the ***describe pod*** *output:*

Controlled By: ReplicaSet/edgy-grizzly-sonatype-nexus-6d6bf984f

To get a Replicaset details please run:

kubectl describe replicaset edgy-grizzly-sonatype-nexus-6d6bf984f*<*

Here we could see that the ReplicaSet is controlled by the Deployment:

Controlled By: Deployment/edgy-grizzly-sonatype-nexus

K8S: Objects

A **Deployment** controller provides declarative updates for Pods and ReplicaSets. You describe a desired state in a Deployment object, and the Deployment controller changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.

K8S: Objects

To get the Deployment list, please run:

kubectl get deployments<

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
edgy-grizzly-sonatype-nexus	1	1	1	1	3h

To get a Deployment details please run:

kubectl describe deployment edgy-grizzly-sonatype-nexus<

K8S: Objects

Like a Deployment, a **StatefulSet** manages Pods that are based on an identical container spec. **Unlike** a Deployment, a StatefulSet maintains a **sticky identity** for each of their Pods. These pods are created from the same spec, but are **not interchangeable**: each has a persistent identifier that it maintains across any rescheduling.

A StatefulSet operates under the same pattern as any other Controller. You define your desired state in a StatefulSet object, and the StatefulSet controller makes any necessary updates to get there from the current state.

K8S: Objects

StatefulSets are valuable for applications that require one or more of the following:

- Stable, unique network identifiers.
- Stable, persistent storage.
- Ordered, graceful deployment and scaling.
- Ordered, graceful deletion and termination.
- Ordered, automated rolling updates.

K8S: Objects

Deployment and Scaling Guarantees:

- For a StatefulSet with N replicas, when Pods are being deployed, they are created sequentially, in order from {0..N-1}.
- When Pods are being deleted, they are terminated in reverse order, from {N-1..0}.
- Before a scaling operation is applied to a Pod, all of its predecessors must be Running and Ready.
- Before a Pod is terminated, all of its successors must be completely shutdown.

K8S: Objects

A **DaemonSet** ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created:

- running a cluster storage daemon, such as glusterd, ceph, on each node.
- running a logs collection daemon on every node, such as fluentd or logstash.
- running a node monitoring daemon on every node, such as Prometheus Node Exporter, collectd, Datadog agent, New Relic agent, or Ganglia gmond.

K8S: Objects

A **Job** creates one or more pods and ensures that a specified number of them successfully terminate. As pods successfully complete, the *job* tracks the successful completions. When a specified number of successful completions is reached, the job itself is complete. Deleting a Job will cleanup the pods it created.

A simple case is to create one Job object in order to reliably run one Pod to completion. The Job object will start a new Pod if the first pod fails or is deleted (for example due to a node hardware failure or a node reboot).

K8S: First deployment

It's high time to create our first deployment!

Using vi (vim) create the deployment yaml file in the user home directory. As the example below, the deployment name is time-tracker-deployment.yaml and application name is time-tracker, please copy-paste the structure and enrich within your's credentials :

```
>vi /home/centos/time-tracker-deployment.yaml
```

_* - please put your docker hub name from the Docker exercises instead of <YOUR_DOCKER_HUB_NAME> placeholder_

K8S: First deployment

```
rsion: apps/v1
kind: Deployment
metadata:
name: time-tracker
labels:
app: time-tracker
spec:
replicas: 3
selector:
matchLabels:
app: time-tracker
template:
metadata:
labels:
app: time-tracker
spec:
containers:
- name: time-tracker
image: <YOUR_DOCKER_HUB_NAME>/time-tracker:latest
```

K8S: First deployment

In this example:

- A Deployment named **time-tracker** is created, indicated by the metadata: name field.
- The Deployment creates three replicated Pods, indicated by the replicas field.
- The selector field defines how the Deployment finds which Pods to manage. In this case, we simply select on one label defined in the Pod template (app: time-tracker).
- The Pod template’s specification, or template: spec field, indicates that the Pods run one container, time-tracker, which runs the time-tracker Docker Hub image at latest version.
- The Deployment opens tomcat’s port 8080 for use by the Pods.

K8S: First deployment

```
>kubectl create -f ~/time-tracker-deployment.yaml
```

Check the deployment, replicaset and pods:

```
>kubectl get deployments
>kubectl get replicasets
>kubectl get pods
```

You should see the 3 pods time-tracker- running there*

K8S: First deployment

*Since the set of our services needs to be published to the outside, let’s create a **LoadBalancer** type service for that. It should create an AWS load balancer to map our 8080 port from all of 3 time-tracker nodes within the 80 port outside:*

>vi /home/centos/time-tracker-LB-service.yaml

K8S: First deployment

```
apiVersion: v1
kind: Service
metadata:
name: time-tracker
labels:
app: time-tracker
spec:
type: LoadBalancer
ports:
- port: 80
targetPort: 8080
protocol: TCP
selector:
app: time-tracker
```

K8S: First deployment

:In this example

- This specification will create a new Service object named “**time-tracker**” which targets TCP port 8080 on any Pod with the "app=**time-tracker**" label to the external targetPort (By default the targetPort will be set to the same value as the port field).
- This Service will also be assigned an AWS load balancer.(type: **LoadBalancer**)
- The Service’s selector will be evaluated continuously and the results will be POSTed to an Endpoints object also named “time-tracker”.
- Kubernetes Services support TCP and UDP for protocols. The default is TCP.

K8S: First deployment

```
>kubectl create -f ~/time-tracker-LB-service.yaml
```

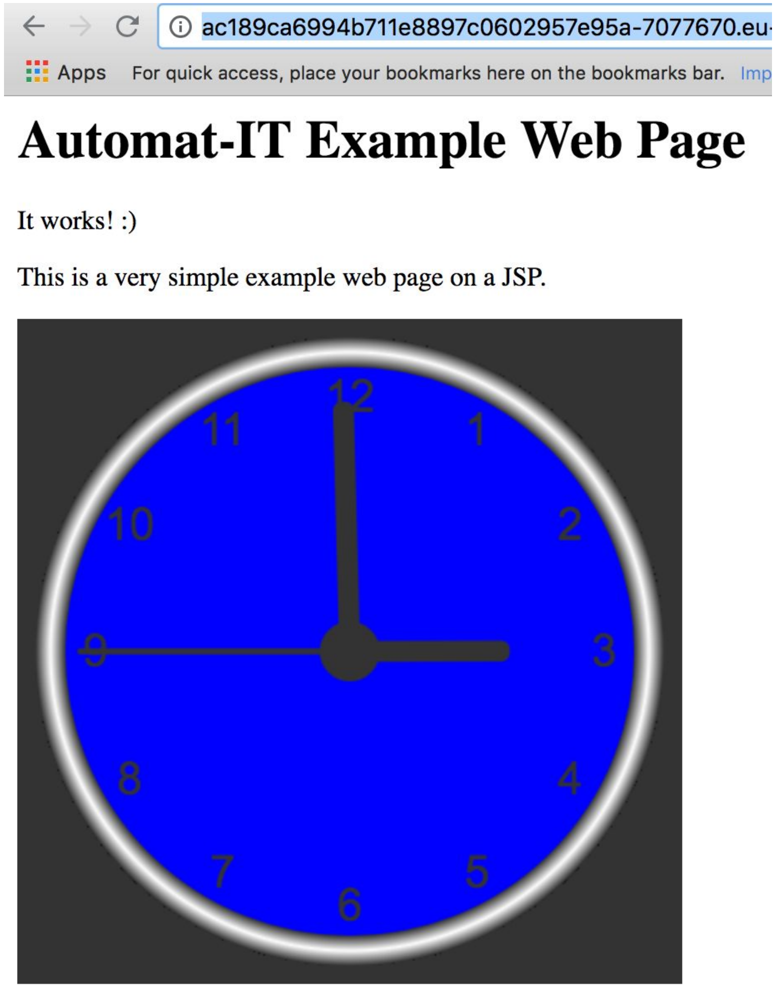
Check the services:

```
>kubectl get svc –o wide
…
time-tracker LoadBalancer 10.233.43.84
ac189ca6994b711e8897c0602957e95a-7077670.eu-central-1.elb.amazonaws.com
80:32076/TCP 2m app=time-tracker
…
```

Go to:

http://**ac189ca6994b711e8897c0602957e95a-7077670.eu-central-1.elb.amazonaws.com**/time-tracker-web-0.3.1/

K8S: First deployment



ac189ca6994b711e8897c0602957e95a-7077670.eu-

Apps
For quick access, place your bookmarks here on the bookmarks bar.
Imp

Automat-IT Example Web Page

It works! :)

This is a very simple example web page on a JSP.

12
1
2
3
4
5
6
7
8
9
10
11

K8S: First deployment

Let’s clean everything up:

```
>kubectl delete deployment time-tracker
>kubectl delete svc time-tracker
```

K8S: Helm Charts

Helm is the **package manager** (analogous to yum and apt) and **Charts** are **packages** (analogous to debs and rpms). The home for these Charts is the Kubernetes Charts repository which provides continuous integration for pull requests, as well as automated releases of Charts in the master branch.

K8S: Helm Charts

:To initiate the helm, please run

cd ~; helm init<

```
Creating /home/centos/.helm
Creating /home/centos/.helm/repository
Creating /home/centos/.helm/repository/cache
Creating /home/centos/.helm/repository/local
Creating /home/centos/.helm/plugins
Creating /home/centos/.helm/starters
Creating /home/centos/.helm/cache/archive
Creating /home/centos/.helm/repository/repositories.yaml
Adding stable repo with URL: https://kubernetes-charts.storage.googleapis.com
Adding local repo with URL: http://127.0.0.1:8879/charts
.HELM_HOME has been configured at /home/centos/.helm$
.Warning: Tiller is already installed in the cluster
(.Use --client-only to suppress this message, or --upgrade to upgrade Tiller to the current version)
!Happy Helming
```

K8S: Helm Charts

:To list all the available helm modules

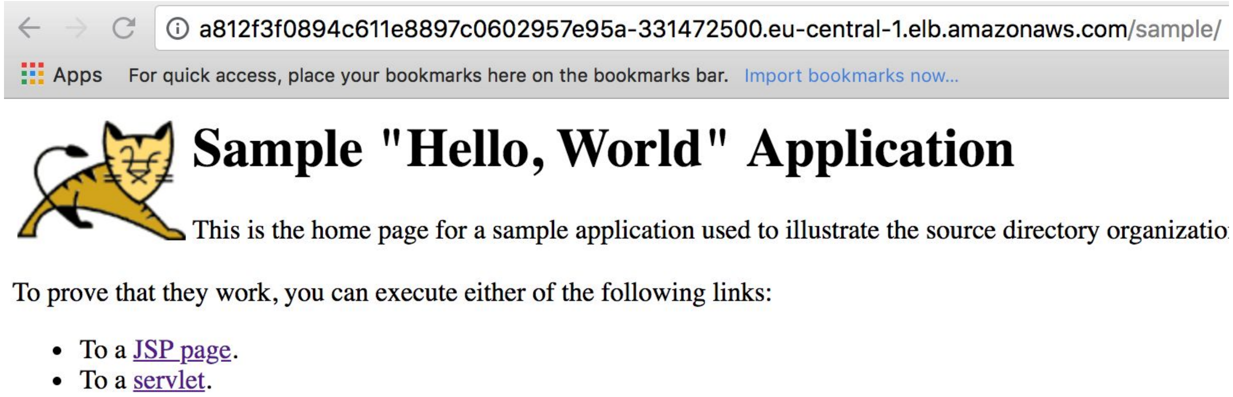
```
helm search<
```

:To install tomcat webserver please run

```
helm install stable/tomcat<
```

:To get the external LB

```
'{"kubectl get svc -o wide | grep tomcat | awk '{print "http://"$4"/sample<
```



a812f3f0894c611e8897c0602957e95a-331472500.eu-central-1.elb.amazonaws.com/sample/

Apps For quick access, place your bookmarks here on the bookmarks bar. Import bookmarks now...

Sample "Hello, World" Application

This is the home page for a sample application used to illustrate the source directory organizatio

To prove that they work, you can execute either of the following links:

- To a JSP page.
- To a servlet.

K8S: Helm Charts

:To delete the tomcat installed by helm

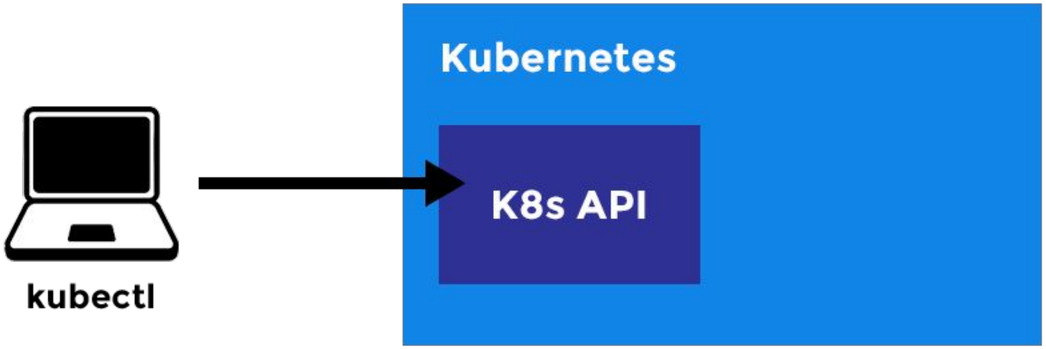
```
’{ kubectl get pods | grep tomcat | awk -F '-' '{print $1"-"$2<
      virtuous-peacock#
```

```
<helm delete <release<
      helm delete virtuous-peacock#
      release "virtuous-peacock" deleted#
```

K8S: Configure kubectl

During our course we used to have the **‘kubectl’** utility as given, however there are cases when you do need to configure it out of the scratch to be able to connect your cluster/clusters from the external environments.

For our main exercise, there is a need to install the kubectl utility on the CI server and connect it with the Kubernetes cluster. This is the real job for the true DevOps like we are!



K8S: Configure kubectl

Please ask tutor with the relevant credentials and connect to the **Ubuntu** CI :server via the SSH CLI. Install the kubectl to the Ubuntu

```
sudo apt-get update && sudo apt-get install -y apt-transport-https<

– curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add<

sudo touch /etc/apt/sources.list.d/kubernetes.list<

echo "deb http://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee -a<
/etc/apt/sources.list.d/kubernetes.list

sudo apt-get update<

sudo apt-get install -y kubectl<
```

K8S: Configure kubectl

:(On the Centos K8S server via the SSH CLI (create a system user+bind

vi ~/account.yml<

```
apiVersion: v1
kind: ServiceAccount
:metadata
name: admin-user
namespace: default
```

vi ~/binding.yml<

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
:metadata
name: admin-user
:roleRef
apiGroup: rbac.authorization.k8s.io
kind: ClusterRole
name: cluster-admin
:subjects
kind: ServiceAccount -
name: admin-user
namespace: default
```

K8S: Configure kubectl

:(On the Centos K8S server via the SSH CLI (create a system user+bind

kubectl create -f ~/account.yml<

kubectl create -f ~/binding.yml <

K8S: Configure kubectl

he Centos K8S server via the SSH CLI

```
kubectl -n default describe secret $(kubectl -n default get secret | grep admin-user<
(’{| awk '{print $1
```

```
Name:         admin-user-token-cxjkh
Namespace:    default
<Labels:      <none
Annotations:  kubernetes.io/service-account.name=admin-user

kubernetes.io/service-account.uid=de954529-94d8-11e8-aba0-0620e18abe3e
Type:  kubernetes.io/service-account-token
…
```

K8S: Configure kubectl

:On the Centos K8S server via the SSH CLI

vi ~/get_conf.sh<

```
server=https://35.159.54.196:6443 #Your Kubernetes Master node external IP
name=admin-user-token-cxjkh #Your tocken from the previous page
('{ca=$(kubectl get secret/$name -o jsonpath='{.data.ca\.crt
(token=$(kubectl get secret/$name -o jsonpath='{.data.token}' | base64 -d
(namespace=$(kubectl get secret/$name -o jsonpath='{.data.namespace}' | base64 -d
" echo
apiVersion: v1
kind: Config
:clusters
name: default-cluster -
:cluster
{certificate-authority-data: ${ca
{server: ${server
:contexts
name: default-context -
:context
cluster: default-cluster
namespace: default
user: default-user
current-context: default-context
:users
name: default-user -
:user
{token: ${token
config < "
```

K8S: Configure kubectl

he Centos K8S server via the SSH CLI

```
chmod +x  ~/get_conf.sh<
get_conf.sh/~<
cat ~/config<
```

Copy the **Centos ~/config** file content to your **Ubuntu: vi ~/.kube/config**

K8S: Configure kubectl

:Check the connectivity from the Ubuntu CI server

```
ubuntu@ip-172-31-44-83:~/.kube$ kubectl --insecure-skip-tls-verify get pods
```

NAME	READY	STATUS	RESTARTS	AGE
edgy-grizzly-sonatype-nexus-6d6bf984f-stbxb	1/1	Running	0	25m

Now we can manipulate with our kubernetes cluster from the external CI !!!environment

Thanks!!!