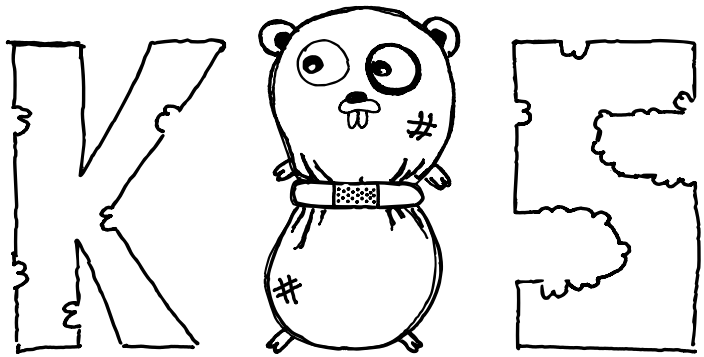


(Kubernetes - K8S)

## Index

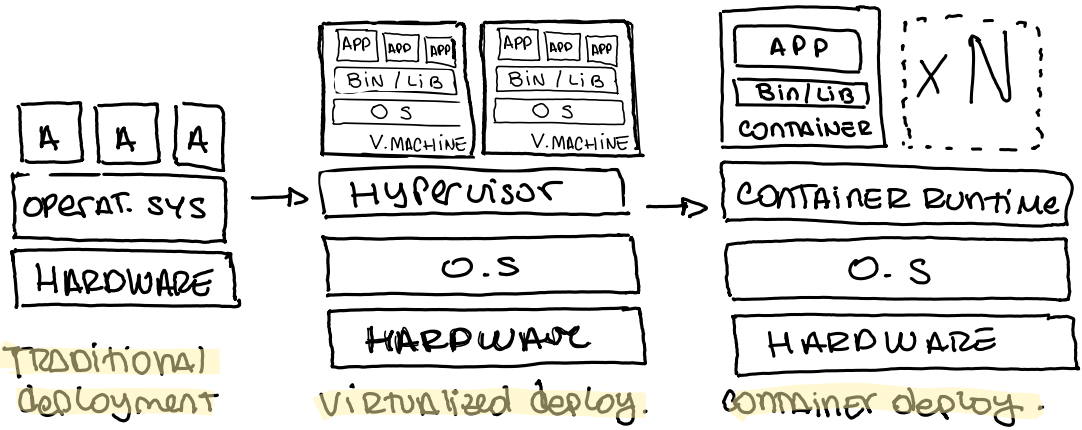


by Lucas Contreras

# Kubernetes (K8S)

**What's K8S:** Open-source container orchestration system for containers. It automates deployment, scaling & management.

## Containarization 101




Traditional deployment — Virtualized deploy. — Container deploy.

## What does K8S Provide:

You need to know what it provides if you wish to know where to aim...

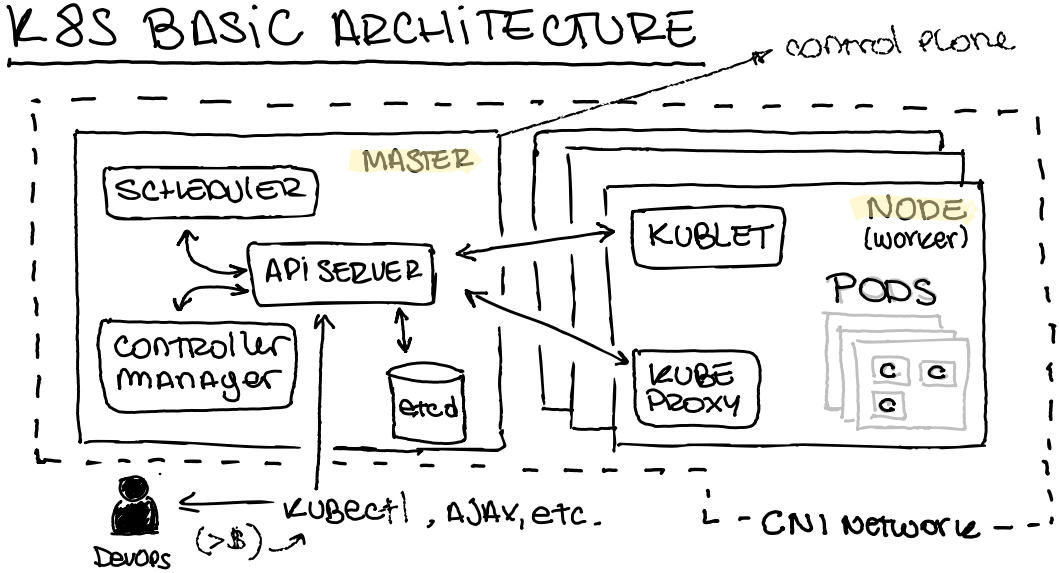
- Service discovery & load balancing.
- Storage orchestration
- Automated rollouts n rollbacks → #SETUP
- Automated bin packing
- Self healing
- secret and config. management.

It's important to know what does K8S provides and what it does not, this lets you forsee any possible problems you think K8S will solved when it actually doesn't. Besides you might find that already addressed issues will reappear since you are changing the whole ARCH.

# Components

Before describing each component we'll take a look to the K8S ARCHITECTURE.

## K8S BASIC ARCHITECTURE




## MASTER COMPONENTS

**Kube-apiserver**: is the front-end for the K8S control plane. It's design to scale horizontally (by deploying more instances).

**Etcd**: consistent and highly-available key value store used as K8S' backing store for all the cluster data.

**Kube-scheduler**: Watches newly created PODS that have no node assigned and selects a node for them to run.

**Kube-controller-manager**: Runs controllers (A control loop that watches the state of the cluster)

Controllers: (they all run under the same process)

- NODE
- REPLICATION
- ENDPOINTS
- SERVICE Account & TOKENS

K-1.01

# NODE COMPONENTS

**Kubelet:** is the K8S agent that runs on each node. It makes sure that containers are running in the pods but it doesn't interfere with the other containers.

**Kube-proxy:** network proxy that runs on each node in the cluster implementing part of the K8S #Service concept. Allows network comunication to your PODS from inside or outside the cluster.

**Container-runtime:** software responsible for running the containers (i.e Docker). K8S supports several container Runtimes.

## The Kubernetes API (introducing kubectl)

This is where we are going to manage our cluster. We are going to do this mostly from a command-line tool called **kubectl** with which we'll be able to create, update, delete and get our API objects. By making calls to the API

## Kubernetes Objects

K-2.0

Just for you not to panic and to get familiar with K8S obj. these are the guys that you'll be able to express in a .yml file

Kubernetes Objects are persistent entities in the K8S system These entities represent the state of your cluster. Specifically:

+ What containarized apps you are running (and on which nodes)
+ The resources available to those applications.
+ The policies arround how those apps behaves (ie. restart...)

A K8S object is a "record of intent" - once the object is created. K8S will always ensure that objects ∃. By creating an object you are telling K8S what you want your cluster to look like.
TO CRUD objects in K8S we use the K8S API, as we mentioned early the kubectl cli makes those API calls for us.

(Too Proteges?)

## Object Specs n' STATUS

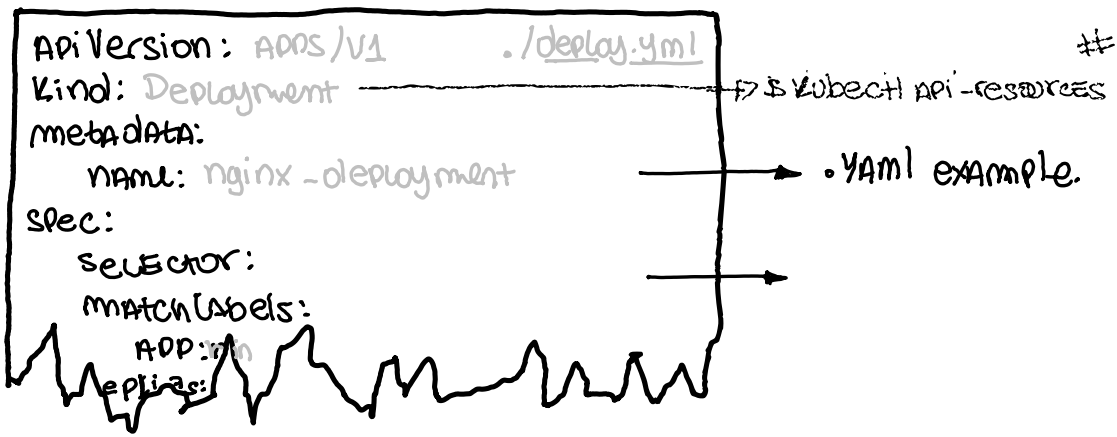
EVERY object includes two nested "object fields" that governs the object configuration: SPECS And STATUS

The **SPEC**, which **YOU** must provide describe the desire state for the object.

The **STATUS** describes the ACTUAL STATE OF THE OBJECT. This last is supplied and updated by **Kubernetes**.

#$> kubectl api-resources → List All Kubernetes objects

When creating an object you send the specs in a JSON format. (Most often you provide this information to kubectl in a .yml file.)



# Kubernetes Object Management.

There're several management techniques and the kubectl supports different way to create and manage K8S objects.

**#WARNING**: A K8S object should be managed using only ONE technique. Mixing and matching technique for the same object may result in undefined behavior.

## A WONDERFUL TABLE I CAME THROUGH

| MANAGEMENT TECHN | OPERATES ON | RECC. ENV | SW | LEARNING CURVE |
|---|---|---|---|---|
| Imperative commands | Live Objects | Develop. | 1+ | Lowest. |
| Imperative object. conf. | Individual files | Production | 1 | Moderate |
| Declarative obj. conf. | Directories | Production | 1+ | Highest |

### Imperative commands.

When using imperatives commands the user operates directly on live obj. of a clusterpassing operation to the kubectl command as arguments or flags.
It provides no history of previous configuration.

**EXAMPLES:** $> kubectl run nginx --image nginx
$> kubectl create deployment nginx --image nginx.

**TRADE-OFFS**: (compared 2 Imp/Dec obj configuration)

**PROs**

- Simple, easy to learn
- Single step to make changes to the cluster

**CONS**

- no template for new obj.
- no integration with change review process
- no source of record
- no audit trail

# Imperative object configuration

In imperatives object conf. commands you specify the operation (create, replace, etc), optional flags and at least one filename. The file must contain a full def. of the object. (YAML or JSON).

#WARNING: the "replace" command replaces the specs with wirne the new one, dropping all changes to the object missing from the conf. file. This approach should not be used with resources whose specs are updated independently from the configuration file.

Examples:

```
kubectl create -f nginx.yml
kubectl delete -f nginx.yml -f redis.yml
kubectl replace -f nginx.yml
```

## Trade-offs:

**Pros** (comp. with Imp. cud)

+ can be stored in Git
+ integrated with reviewing processes.
+ Provides a template.

**Cons** (comp. with Imp. cud)

- requires understa. obj. sch.
- you have to write a YML

**Pros** (comp with Dec. obj conf.)

+ Is simpler and easier to understand
+ As of k8s 1.5, Imp. cud is more mature.

**Cons** (comp with Dec. obj conf)

- doesn't work well on dirs
- Updates to live obj. must be reflected on conf. files or they will be lost on the next replacement.

# Declarative Object Configuration.

When using DOC, the user operates on conf. files stored locally, however the user does not define the operations to be taken on the files. Create, Update and Delete are automatically detected per-object by kubectl. This enables working on directories where ≠ operations might be needed for ≠ objects.

Example:

```
$>kubectl diff -f configs/    (1)
$> kubectl apply -f configs/  (2)
```

(2) Process All conf. files in the configs Directory.
(1) Check changes by applying.

## TRADE-OFFS (compared to Imp. Object configuration)

| PROs | CONs |
|---|---|
| + Changes to live OBJ. are retained, even if not merged to the conf. f. | - Harder to DEBUG and under-stand when results are unexpect. |
| + Better support for directories and auto-detecting oper. types. | - Partial updates using diff create complex merge and PATCH operations |



## Names and UIDs

All Object are unAmbiguously identified by a Name and an UID

Each Object has an unique Name and it's a string specified under the tag "metadata:" in the yaml. file.
# Check Conventions. (max 253 char, lowercase + "-" and ".")

UID are unique strings assigned and managed by K8S

# Namespaces (virtual clusters)

Namespaces is how K8S manage to distinguish between multiple virtual clusters. These virtual clusters ARE called Namespaces

Namespaces are intended for use in env. 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.

- Namespaces provide a scope for Names (K-2.2)
- Namespaces can not be nested inside one another
- Each K8S resource can be in only ONE Namespace

## Default Namespaces

**default** → where resources with no NS. go.
**kube-system** → NS for K8S resources.
**kube-public** → readable by all users. mostly for cluster usage

## Labels and Selectors.



Label are key/value pairs attached to obj. below the metadata tag. They are used to organize and select a subset of objects.

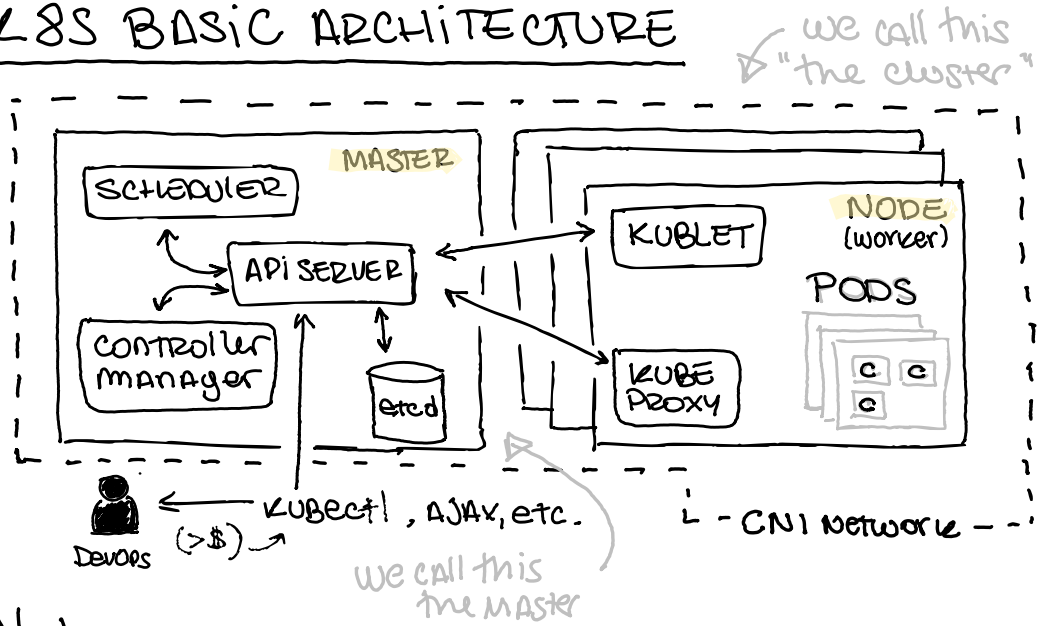
#Unlike name and UID label are not unique

- There are two types of Selector: Equality-based and Set-based
- Both labels and selectors are specified on the YML file or via command line. (with the flag -l)

(For + info on K8S please read about Anotations, Field selectors)

# Kubernetes Architecture.

## K8S BASIC ARCHITECTURE




**Nodes**: A node is a worker machine.
A node may be a VM or a bare metal machine
Each node contains the services necessary to run PODs
And is managed by the master components.
Services in a node (worker) includes

- container runtime.
- kube-proxy
- kubelet

} previously defined components.

**Node Status** A node STATUS contains the following info.

- **Addresses** (Hostname, InternalIP, ExternalIP)
- **Conditions** (OutOfDisk, Ready, MemoryPressure, PIDPressure, ...)
- **Capacity and Allocatable** (Describes CPU, Mem & max PODs Avb.)
- **Info.** (Gral info about the node)

# Management

Unlike PODS and services nodes are not created by K8S. They are created by external cloud providers or they are on your pool of physical or virtual machines.

Nevertheless, what K8S does is, it creates a node (internally) and then checks if that node exists. like.

```
{ "kind": "Node",
  "apiVersion": "v1"
  "metadata": {
      "name": "10.240.79.167",
      ...
  }
}
```

K8S will check if the node is based on the [metadata.name]

# K8S will keep the node created as "invalid" and will keep on performing healthchecks on the node until it becomes valid.

**Node controller.** The node controller is a K8S master component. which manages various aspects of nodes it has 3 main roles on a Node lifes.

1. ASSING A CIDR block to the node when it is registered.
2. Keep the nodes controller's list of nodes up to date with the cloud provider's list of available machines. Wherever a node is unhealthy is the node-controller the one who asks the cloud provider if the VM is still available.
3. Monitoring node's health. it updates the Node Ready condition on the NodeStatus

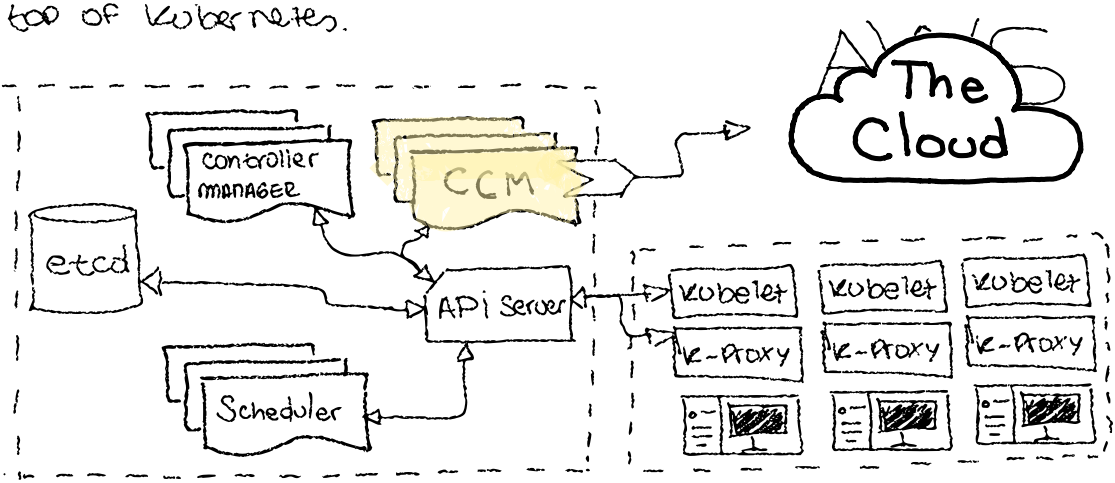
**Self-registration of Nodes:** (Nodes can be manually admin.)

- The kubelet comp. will register itself with the API unless the flag --register-node is set to FALSE explicitely

# The Cloud controller Manager

The CCM was originally created to allow specific vendor code and the K8S core to evolve independent of one another.

The CCM runs alongside the MASTER components such as the controller manager, API server and scheduler. it can also be started as a Add-on in which case it runs on top of kubernetes.




For more info on the CCM Please refer to the K8S site.

## Vendor implementations of the CCM

- AWS
- Azure
- Oracle :/
- Digital Ocean
- GCP
- Linode
- Baidu Cloud
- (Google Cloud ?)

# POD "THE SMALLEST DEPLOYABLE OBJ."

**A POD**: IS A GROUP OF ONE OR MORE CONTAINERS.

- A POD IS THE BASIC EXECUTION UNIT OF K8S APPS.
- A POD represents PROCESSES running on your cluster.
- A POD encapsulates an APPLICATION'S container (or more) storage resources, A unique network IP and options on how those containers should RUN.

TWO main way of usage.

**RUN A SINGLE CONTAINER**: The "one-container-per-POD" model is the most common K8S USE CASE. YOU CAN think A POD AS A WRAPPER FOR A single container.

**RUN MULTIPLE CONTAINERS THAT NEEDS TO WORK TOGETHER**: A POD might contain an APP. that consist of multiple CO-LOCATED containers that Are tightly coupled And need to share resources.

EACH POD is meant TO RUN A **SINGLE INSTANCE OF AN APP.** IF YOU WISH TO **SCALE HORIZONTALLY** you would like to have SEVERAL PODs. In K8S, THESE ARE known AS **REPLICAS.**

POD's Shared resources → Networking.
→ Storage.

#PODS are designed AS relatively ephemeral, disposable entities.

# PODs DO not run by themself, is an environment where the containers run in and persist.

# Pods and Controllers.

A controller can create and manage multiple PODs for you. For example, if a node fails, the Controller can replace the POD by scheduelling an identical one on a different node.
**Deployments, StatefulSet and DaemonSet** are some example of controllers that contain one or more PODs.

**Pod templates.** are POD specification (.yml.) which are included in another objects.

./Pod.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: myAPP-Pod
spec:
  containers:
  - name: myAPP-container
    image: busybox
    command: ["sh ..."]
```


FOR FURTHER READING ON PODS I DO SUGGEST YOU TO READ ABOUT:

- Pod conditions → this could help.
- RESTART POLICY → THESE ARE A MUST
- CONTAINER STATES → THESE ARE A MUST
- INIT CONTAINERS → this is interesting
- POD PRESETS → this is interesting

# Controllers

A controller is a control loop that ensures the desire state of the cluster (PODs) is the initially specified.
As we mentioned before controllers are responsable for PODS. In this section we are going to list some main controllers and how they work.

## ReplicaSets (before learning controllers we need to understand ReplicaSets)

A ReplicaSet purpose is to maintain a stable set of replica PODs at a given time. Is often use to guarantee the availability of a specific number of identical PODs.

A replicaset ensure the availability of PODs. However, a Deployment is a higher-level concept that manages ReplicaSets and do more. K8S, Recommends to use Deployments instead of using ReplicaSets directly unless you require custom update orchestration. or don't require updates at all.

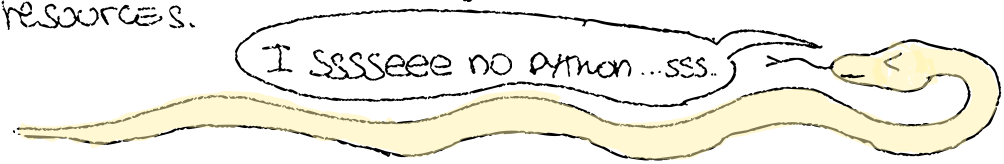
## Deployments

#NOTE: A "Deployment" that configures a "ReplicaSet" is now the way to set up Replication.

K-2.1

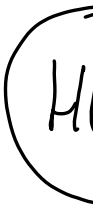
A Deployment controller provides declaratives updates for PODs and ReplicaSets.

You can use Deployments to describe a desire state, create new ReplicaSet or remove 3ing Deployments and adopt all its resources.



# Use cases for deployments:

+ to rollout a replicaset ——→ There's a gr8 example on the k8s site to get started with.
+ declare new state of the PODs
+ scale up a deployment
+ pause the deployment
+ clean replicasets.



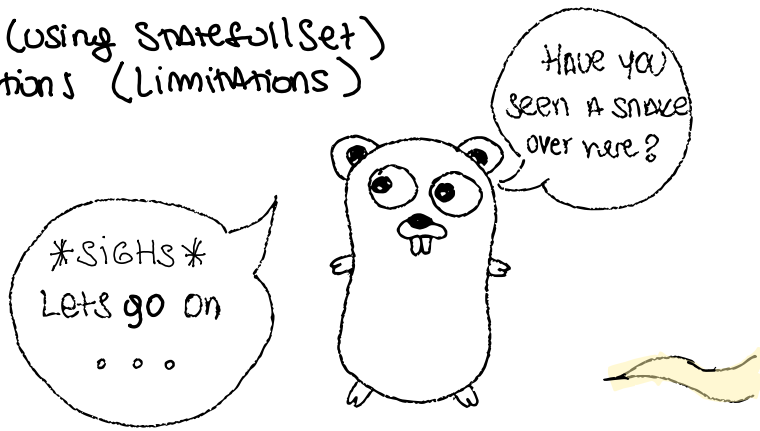

# StatefullSet

StatefullSet is the workload API obj used to manage stateful applications.

Like Deployments, StatefulSets manage PODs that are based on identical container specs. Unlike Deployment, it maintains a sticky identity for each of their pods.
These pods are created from the same specs, but are not interchangeable: each has a unique identifier that maintains across any rescheduling.

For further reading on StatefullSets please consider reading following concepts from the k8s site:

- StatefullSet usage (using StatefullSet)
- StatefullSet Limitations (Limitations)
- Pod Identity.



# DaemonSets

A DaemonSet ensures that all (or some) nodes run a copy of a POD. As nodes are added to the cluster, PODs are added. As nodes are removed from the cluster the PODs are garbage collected.

Use cases are.

+ Running a cluster storage daemon, such as glusterd or ceph on each node.
+ Running logs collection daemon on every node, such as fluentd or logstash.
+ Running a node monitoring daemon on every node such as: Prometheus Node Exporter, Sysdig Agent, collectd, gmond, NewRelic Agent

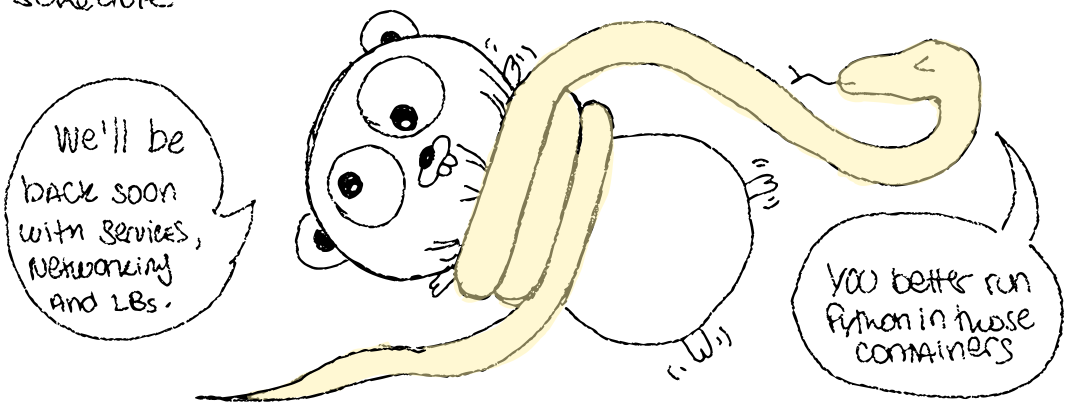
## Jobs

A Job controller creates one or more PODs and ensures that a specifically number of them are terminated successfully. When a specified number of successful completions is reached the Job is complete.

## Cronjob.

Schedule

A Cron Job creates a Job on a time-based



V-6.0

# Services

A service is what you wanna read about when you are trying to expose an app that running on a set of PODs as a network service.

K8S Give PODs their own IP address and a single DNS for a set of PODs, you can of course load balance across them.

To set a set of Pods to be targeted by a Service you use Selectors (k-l.4) but Services ∃ independently from selectors.

## Define a service. (guess what? It's an Object too)

Suppose you have a set of PODs that each carry the label app=My-App and they all listen to port 9376 TCP.

./svc.yml

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: My-App
  ports:
  - protocol : TCP
    port: 80
    targetPort: 9376
```

## Services without Selectors

Provide abstract access to K8S, here's why you want to do that:

- X You want to have an external db in Prod, but your test env uses your own db.
- X You want to point your Svc to a # Svc on another NameSp. or to another cluster.
- X Migration purposes.

# Publishing Services (ServiceTypes)

We are not going to describe each type in particular nor explain how they work. For more info on that please refer to the DOC

# If Not explicitely specified, the default type is **ClusterIP**

**ClusterIP**: Exposes the service on a cluster-internal IP. Choosing this value makes the service only reachable from within the cluster.

**NodePort**: Exposes the Service on each Node's IP at a static port (the NodePort). A **ClusterIP** service to which the nodeport routes, is created automatically. You will be able to contact the service from outside the cluster by requesting **<Node's IP>:<NodePort>**

**LoadBalancer**: Exposes the service externally using a cloud provider's load balancer **NodePort** & **ClusterIP** are created automatically.

**ExternalName**: Maps the service to the content of the External name field. (e.g. foo.bar.example.com) by returning a CNAME record with its value. No proxing of any kind is setup.

## DNS

Every service define in the cluster gets a DNS name

Assume a service named `foo` in the K8S Namespace `bar`. A POD running under under `bar` can look up this service by "`foo`" A POD running in a Namespace `quxx` can find this service by "`foo.bar`"