**What is Kubernetes**

* Kubernetes is an open source **orchestration** system for Docker containers
  + It lets you schedule **containers** on a cluster of machines
  + You can run **multiple containers** on one machine and those machines then make up a **cluster**
  + You can run long running **services** (like web applications) on your kubernetes cluster
  + Kubernetes can **manage** the state of these containers
    - Can start the containers on specific nodes
    - Will restart the container when it gets killed
    - Can move containers from one node to another. For instance if are going to do maintenance on one node, Kubernetes can take all the containers from one node and move them to the other node.
* Instead of just running a few Docker containers on one host manually, Kubernetes is a platform that will manage the containers for you
* Kubernetes cluster can start with one node and can go up to until thousands of nodes
* Some other popular container orchestrators are:
  + **Docker Swarm** – which comes with Docker by default but is not as extensive as Kubernetes
  + **Mesos**

**Kubernetes Advantages**

* You can run **Kubernetes** anywhere:
  + On premise (own Datacenter)
  + Public (Google Cloud, AWS, MS Azure)
  + Hybrid: Public and Private
* **Highly modular** – so you can make changes if necessary
* **Open Source** – The project is on GitHub and you can easily make changes
* **Great Community** – The Kubernetes team encourages everyone who joins the community to make changes which makes it a great community
* **Backed by Google** – The initial version was being released by Google and they still support it and it incorporates all the good stuff that Google had done over the decades

**Kubernetes Setup**

* **Kubernetes** should really be able to run **anywhere**
* But, there are more **integrations** forCloud Providers like AWS & GCE
  + Things like **Volumes** and **External Load Balancers** work only with **supported** cloud providers
* I will first use **minikube** to quickly spin up a local single machine with a Kubernetes cluster
* We also can spin up a cluster on AWS using **Kops**
  + Kops is a tool which can be used to spin up a **highly available production cluster** on AWS

**Checking Minikube Setup**

* **kubectl run hello-minikube --image=gcr.io/google\_containers/echoserver:1.4 --port=8080**
* **kubectl expose deployment hello-minikube --type=NodePort**
* **minikube service hello-minikube --url**

**Running First App On Kubernetes**

* Let’s run our newly built application on the new Kubernetes cluster
* Before we can launch a container based on the image, we need to create a **pod definition**
  + **A pod** describes an application running on Kubernetes
  + A pod can contain **one or more tightly coupled containers**, that make up the app
    - Those apps can easily communicate with each other using their local **port numbers**
  + Our app only has one container
* Create a file pod - helloworld.yml with the pod definition:



* Use kubectl to create the pod on the Kubernetes cluster:
* **kubectl create –f first-app/helloworld.yml**
* **kubectl describe pod nodehellorbc.example.com**
* **kubectl port-forward nodehellorbc.example.com 8081:8000** **This is the one way**
* **kubectl expose pod nodehellorbc.example.com --type=NodePort --name=nodehellorbc-service This is the second way**
* **minikube service nodehellorbc-service --url**

**Useful Commands**

* **Kubectl get pod Get information about all running containers**
* **Kubectl describe pod <pod> Describe one pod**
* **Kubectl expose pod <pod> --port=444 –name=frontend Expose the port of a pod (Creates new service)**
* **Kubectl port-forward <pod> 8080 Port forward the exposed pod port to your local machine**
* **Kubectl attach <podname> -i Attach to the pod**
* **Kubectl exec <pod> -- command Execute a command on the pod**
* **Kubectl label pods <pod> mylabel=xyz Add a new label to the pod**
* **Kubectl run –i –tty busybox –image=busybox --restart=Never -- sh Run a shell in a pod – very useful for debugging**

**Replication Controller**

* Scaling in Kubernetes can be done using the **Replication Controller**
* The replication controller will **ensure** a specified number of **pod replicas** will run at all time
* A pods created with the replication controller will **automatically** be **replaced** if they fail, get deleted or are terminated
* Using the RC is also **recommended** if you just want to make sure **1 pod** is always running, even after reboots
  + You can then run a RC with just **1 replica**
  + This make sure that the pod is always running
* **Kubectl create –f replication-controller\helloworld-repl-controller.yml**
* **Kubectl get pods**
* **Kubectl get rc**
* **Kubectl describe pod <podname>**
* **Kubectl scale --replicas=4 -f replication-controller\helloworld-repl-controller.yml Using file name**
* **Kubectl scale --replicas=1 rc/<replication-controller name> Using rc name**

**Replication Set**

* **Replication Set** is the next-generation Replication Controller
* It supports a new selector that can do selection based on **filtering** according a **set of values**
  + e.g. “environment” either “dev” or “qa”
  + not only based on equality, like the Replication Controller
    - for ex. In RC you can only say “environment” == “dev” and you couldn’t do anything more complex. With Replication Set you can do more complex matching
* This **Replica Set**, rather than Replication Controller, is used by the Deployment object.

**Deployments**

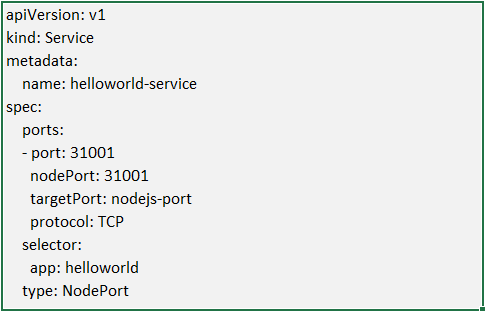
* A deployment declaration in kubernetes allows you to do app **deployments** and **updates**
* When define the deployment object, you define the **state** of your application
  + The Kubernetes will then make sure that the cluster matches your **desired** state
* Just using **RC** and **Replication Set** might be cumbersome to deploy apps – So these 2 things might not always be fit to deploy your apps. It would be too much manual work to then do updates and so on.
  + The **Deployment Object** is easier to use and gives more possibilities. And following are those possibilities
  + With the Deployment Object you can:
    - **Create** a deployment (e.g. deploying an app)
    - **Update** a deployment (e.g. deploying a new version)
    - Do **rolling updates** (zero downtime deployments)
    - **Rollback** to a previous version
    - **Pause/Resume** a deployment (e.g. to rollout to only a certain percentage of your running pods)

**Useful Commands**

* **Kubectl create –f deployment/helloworld.yml Create the deployment**
* **Kubectl get deployments**
* **Kubectl get rs**
* **Kubectl get pods –show-labels**
* **Kubectl rollout status deployment/hellorbc-deployment**
* **Kubectl expose deployment hellorbc-deployment --type=NodePort**
* **Kubectl get service**
* **Minikube service hellorbc-deployment --url**
* **Kubectl set image deployment/hellorbc-deployment kmlchauhan/hellorbc:1.2**
* **Kubectl edit deployment/hellorbc-deployment** **Edit the deployment object**
* **Kubectl rollout status**
* **Kubectl rollout history deployment/hellorbc-deployment**
* **Kubectl rollout undo deployment/hellorbc-deployment**
* **Kubectl rollout undo deployment/hellorbc-deployment --to-revision=n**

**Services**

* **Pods** are very **dynamic**, they come and go on the Kubernetes cluster.
  + When using a **Replication Controller**, pods are **terminated** and created during scaling operations
  + When using **Deployments**, when **updating** the image version, pods are **terminated** and new pods take the place of older pods
* That’s why pods should never be accessed directly, but always through a **Service**
* A Service is the **logical bridge** between the “mortal” pods and other **services** or **end-users**
* When using the “kubectl expose” command earlier, we created a new Service for our pod, so it could be accessed externally.
* Creating a service will create an endpoint for our pod(s):
  + a **ClusterIP**: a virtual IP address only reachable from within the cluster (this is the default)
  + a **NodePort**: a port that is the same on each node that is also reachable externally
  + a **LoadBalancer**: a LoadBalancer created by the cloud provider that will route external traffic to every node on the NodePort (ELB on AWS)
* The options just shown only allow you to create **virtual IPs** or **ports**
* There is also possibility to use **DNS names**
  + **ExternalName** can provide a DNS name for the service
  + e.g. for service discovery using DNS
  + This only works when the **DNS add-on** is enabled
* This is an example of a Service definition (also created using kubectl expose)



* Note: by default service can only run between ports 30000-32767, but you could change this behavior by adding the –service-node-port-range=argument to the kube-apiserver (in the init scripts)

**Labels**

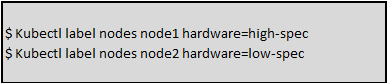
* Labels are key/value pairs that can be attached to objects
  + Labels are like **tags** in AWS or other cloud providers, used to tag resources
* You can **label** your **objects**, for example your pod, following an organizational structure
  + **Key**: environment – **Value**: dev / staging / qa / prod
  + **Key**: department – Value: engineering / finance / marketing
* Previously also we have used tags in our .yaml file:



* Labels are **not unique** and **multiple labels** can be added to one object
* Once labels are attached to an object, we can use filters to narrow down results
  + This is called **Label Selectors**
* Using Label Selectors, you can use **matching expressions** to match labels
  + For instance, a particular node can run on a node labeled with “environment” equals “development”
  + More complex matching: “environment” in “development” or “qa”

**Node Labels**

* We can also use labels to tag **nodes**
* Once nodes are tagged, you can use **label selectors** to let pods run only on **specific nodes**
* There are **2 steps** required to run a pod on a specific set of nodes:
  + First you **tag** the node
  + Then you add a **nodeSelector** to your pod configuration
* First step, add a label or multiple labels to your nodes:

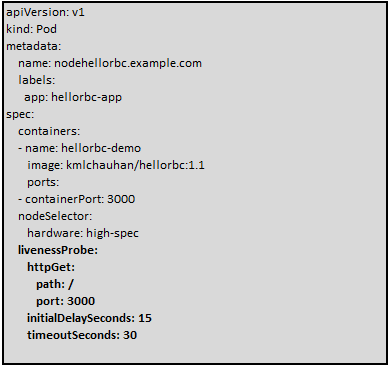


* Secondly, add a pod that uses those labels:



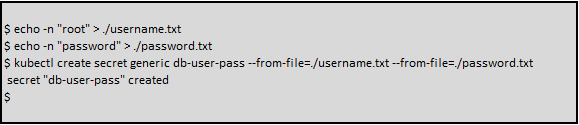
**Helath Checks**

* When you put your application in production you will definitely to configure health checks.
* If your application **malfunctions**, the pod and container can still be running, but the application might not work anymore.
* To **detect** and **resolve** problems with your application, you can run **health checks**
* You can run 2 different type of health checks
  + Running a **command** in the container **periodically**
  + Periodic checks on a **URL** (HTTP)
* The typical production application behind a load balancer should always have **health checks** implemented in some way to ensure **availability** and **resiliency** of the app
* This is how a health check looks like in below sample container:

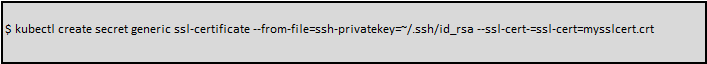


**Secrets**

* Secrets provides a way in Kubernetes to distribute **credentials**, **keys**, **passwords** or “**secret**” **data** to the pods
* Kubernetes itself uses this Secrets mechanism to provide the credentials to access the internal API
* You can also use the **same mechanism** to provide secrets to your application
* Secrets is one way to provide secrets, native to Kubernetes
  + There are still **other ways** your container can get its secrets if don’t want to use Secrets (e.g. using an external vault services in your app)
* Secrets can be used in the following ways:
  + Use secrets as **environment variables**
  + Use secrets **as a file** in a pod
    - This setup uses **volumes** to be mounted in a container
    - In this volume you have **files**
    - Can be used for instance for dotenv files or your app can just read this file
  + Use an **external image** to pull secrets (from a **private image registry**)
* **First you need the secrets,** and to generate secrets using files – you can use below commands:



* A secret can also be an SSH key or an SSL certificate



* If you want to generate the secrets using yaml definitions:
* sss