**Kubernetes**

(From *kubernetes.io*, DigitalOcean’s community tutorials, KodeKloud’s *Kubernetes for the Absolute Beginners*)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Terms** | | | | |
| Kubernetes | - System for running and coordinating containerised applications across a cluster of machines  - Platform designed to completely manage the life cycle of containerised applications using methods that provide predictability, scalability and high availability  - Define how applications are run and their interactions with other applications/outside world  - Scale services up/down  - Perform rolling updates  - Switch traffic between different versions of applications to test features/rollback problematic deployments | | | |
| Summary | - A cluster consists of Master and nodes.  - Nodes runs pods.  - Pods are the basic execution unit and contain containers and shared resources. | | |  |
| Container orchestration | - Deploy and managing containers  - Orchestrate the connectivity and automatically scale up or down based on the load  - Examples of CO technologies: Docker Swarm, Kubernetes, MESOS | | |  |
| Advantages  - Application is highly available (hardware failures are not of much consequence - application running on different nodes)  - User traffic load balanced across various containers (scale up and load accordingly automatically) | | |  |
| **CO Tech** | |  | |
| Docker Swarm | | 😊 Easy to set up and get started  ☹ Lacks some advanced features required for complex applications | |
| Kubernetes | | 😊 Provides a lot of options to customize deployments of complex architectures  ☹ Just a bit difficult to set up | |
| MESOS | | 😊 Supports very advanced features  ☹ Quite difficult to set up | |
| **kubectl** | - Used to deploy and manage applications on a Kubernetes cluster | | |  |
|  |  | | |  |
|  |  | | |  |
|  |  | | |  |
|  |  | | |  |
|  |  | | |  |
|  |  | | |  |
|  |  | | |  |
|  |  | | |  |
| **Code** | | **Explanation** | | |
| kubectl [command] [TYPE] [NAME] [flags] | | Common syntax | |  |
| kubectl [action] [resource] | | - Performs specified action on specified resource |
| create, describe | | **Actions** | |  |
| node, container | | **Resources** | |  |
| kubectl **version** | | See version of client and server, details about build | |  |
| kubectl **get <resource>** | | List resources  - In the current (default) namespace | | Example:  kubectl get services |
|  | | Show detailed information about a resource | |  |
| kubectl **get all** | | List all objects in cluster | |  |
| --help | | Add after commands to get additional info about possible parameters | | Example:  kubectl get nodes --help |
|  | |  | |  |
| curl ifconfig.co/json | | When run on a server, can get exit node information (?) | | Sample return  {“ip”: “88.198.16.116”, “ip\_decimal”:1489375348, “country”: “Germany”, “country\_iso”: “DE”,  “country\_eu”: true, …} |
| docker run -d --name=frontend --link worker | |  | |  |
|  | |  | |  |
|  | |  | |  |

**Cluster, Master, Nodes, Pods, Containers**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **K8s cluster** | | | | | |
| - Highly available cluster of computers coordinated and connected to work as a single unit  - Deploy applications to a cluster without tying them specifically to individual machines (containerisation)  - Distribution, scheduling (placement) and execution of application containers across cluster automated, and done efficiently  - Consists of 2 types of resources:  1) **Master** – coordinates the cluster  2) **Nodes** – workers that run applications  - Cluster that handles production traffic should have a minimum of 3 nodes  - If not, every cluster has at least one worker node | | | |  | |
|  | |  | |  | |
|  | |  | |  | |
| **Code** | | **Explanation** | | | |
| kubectl **cluster-info** | | View cluster details | |  | |
| kubectl **run** [app-name] | | Deploy application on the cluster | |  | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |
| **Master** | | | | | |
| - Responsible for managing cluster and nodes used to host the running applications  - Coordinates all activities in cluster  - Decides how to distribute processes *across* nodes in the cluster  - Examples of activities  - Scheduling applications  - Maintaining applications’ desired state  - Scaling applications  - Rolling out new updates  - Authenticate clients and nodes | | | |  | |
| When applications deployed on Kubernetes, Master:  - Starts application containers  - Schedules containers to run on the cluster’s nodes  - Acts as a “gateway” for the cluster by exposing an API for users and clients | | | |  | |
| Components associated with master server:  **1) *etcd* server**  - Store configuration data used to manage cluster, light-weight key-value store  - Does this in a distributed manner  - Globally available, reliable, can be configured to span across multiple nodes (accessible by all nodes in the cluster)  - Store information about multiple nodes and Masters in cluster  - Implements locks to ensure no conflicts between the Masters  - Used for service discovery and help components configure/reconfigure themselves according to up-to-date information  **2) kube-apiserver (Kubernetes API)**  - API exposed by Master  - Can be used by end users directly interact with the cluster  - Acts as frontend for Kubernetes, to interact with the K8s cluster  **3) *kubelet* service**  - Agent that runs on each node in the cluster, manages them  - Makes sure that containers are running on the nodes as expected  - Communicated with Kubernetes master (receives work instructions)  - Has tools for handling container operations (e.g. *Docker, rkt*)  **4) container runtime**  - Underlying software used to run containers  - E.g. Docker, rkt  **5) a bunch of controllers**  - Responsible for noticing or responding when nodes, containers and end-points go down  - Makes decisions to bring up new containers in such cases  **6) scheduler**  - Responsible for distributing or scheduling work across multiple nodes  - Assigns newly created containers to nodes | | | |  | |
| When applications deployed on Kubernetes,  - Starts application containers  - Schedules containers to run on the cluster’s nodes  - Acts as a “gateway” for the cluster by exposing an API for users and clients | | | |  | |
|  | |  | |  | |
|  | |  | |  | |
| **Nodes** | | | | | |
| - A VM or physical computer that serves as a worker machine (virtual/physical) in a Kubernetes cluster  - Runs pods, each node can have multiple pods (they can share the load)  - Each node managed by a Master, which automatically handles scheduling of the pods in all node in the cluster  - Scheduling takes into account the available resources on each node  - Every Kubernetes node runs at least:  1) Kubelet  2) Container runtime (e.g. Docker, rkt)  - Responsible for pulling container image from registry, unpacking container and running the application  - Containers can only be scheduled together in a single Pod if  - They are tightly coupled  - Need to share resources such as disk | | | |  | |
| When applications deployed on Kubernetes, Nodes:  - Communicate with master using Kubernetes API | | | |  | |
|  | |  | |  | |
|  | |  | |  | |
| Difference between Master and Worker node:  **Worker node**  - Where containers are hosted  - Has the container runtime (e.g. Docker)  - Has the kubelet service that interacts with the kube-apiserver in Master (to provide health information of the worker node and carry out actions requested by Master)  **Master node**  - Has the kube-apiserver (defining characteristic of a Master)  - Stores all information gathered in key-value store etcd  - Also has the controller and scheduler | | | | Note which components are on which kind of node | |
| **Code** | | **Explanation** | | | |
| kubectl **get nodes** | | View nodes in the cluster  - If status READY, node is ready to accept applications for deployment | | Shows  NAME, STATUS, ROLES, AGE, VERSION | |
| kubectl get nodes **-o wide** | | View more details about nodes in cluster | | Shows  NAME, STATUS, ROLES, AGE, VERSION, INTERNAL-IP, EXTERNAL-IP, OS-IMAGE, KERNEL-VERSION, CONTAINER-RUNTIME | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |
| **Pods** | | | | | |
| - The basic execution unit of a Kubernetes application/a single instance of an application  - Created when a Deployment is created in Kubernetes (containers are encapsulated inside pods)  - Each pod is tied to the node where it is scheduled, and remains there until termination/deletion  - According to restart policy  - In case of node failure, identical pods are scheduled on other available nodes in the cluster  - Have a lifecycle: when node dies, pods running on node also lost  - Represents processes running on your cluster/a group of one or more application containers (e.g. Docker, rkt) and some **shared resources** for those containers  - Each pod in a node has a unique IP address  - Shared resources  - Shared storage (e.g. *Volumes*)  - Networking (e.g. unique cluster IP address)  - Information about how to run each container (e.g. container image version/specific ports to use)  - Models an application-specific “logical host”  - Can container different application containers which are relatively tightly coupled  - Might include container with Node.js app and a different container that feeds the data to be published by the Node.js webserver  - Containers in a pod  - Share an IP address and port space  - Always co-located and co-scheduled  - Run in a shared context on the same Node  - A pod always runs on a node  - Pods run in an isolated private network  - Need proxy access to them (so to debug and interact with them) | | | |  | |
| - Usually have a 1-to-1 relationship with containers (i.e. 1 pod, 1 container)  - But a single pod can have multiple containers, though usually not applications of the same kind  - For example, helper container that does supporting tasks (e.g. processing user entered data, file for web application)  - Containers in the same pod can communicate with each other directly by referring to each other as local host since they share the same network space, and possible storage space  - Containers in the same pod will be created and destroyed together  - Pods allow application to be equipped for scaling and architectural changes in the future | | | |  | |
| - Each pod gets its own IP address | | | |  | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |
| **Code** | | **Explanation** | | | |
| kubectl run nginx --image nginx | | Creates a pod with a single container in the node | | At this stage, only accessible internally from the node | |
| kubectl run **--generator**=run-pod/v1 nginx --image=nginx | | Create pod using --generator | | - Use keywords generator and restart to create a pod, else command will create a **deployment** instead (deprecated method, not advised) | |
| kubectl run nginx --image=nginx **--restart=Never** | | Create pod using --restart=Never | |
| kubectl **get pods** | | List all existing Pods  - If no Pods are running, then it means the interactive environment is still reloading its previous state  - Wait a couple of seconds and list the Pods again, continue once you see one Pod running | | Returns  NAME  READY  STATUS  RESTARTS  AGE | |
| **Returns** | | | |
| READY | | Lists the No. of Running Containers/Total No. of Containers | |
| kubectl get pods **-o wide** | | List all existing Pods with more information | | Additional information:  IP, NODE, NOMINATED NODE, READINESS, GATES | |
| kubectl get … -l <label> | | Query list of pods with specified label | |  | |
| kubectl **describe pods** <pod\_name> | | More details about the pod than get  **- View** what **containers** are inside the Pod, what **images** used to build those containers  - Output is designed to be human readable, not to be scripted against | | Returns details about the Pod’s container:  - IP address  - Ports used  - A list of events related to the lifecycle of the Pod | |
| kubectl **exec** | | Execute a command on a container in a pod | |  | |
| echo -e “\n\n\n\e[92mStarting Proxy. After starting it will not output a response. Please click the first Terminal Tab\n”  kubectl proxy | | 1) Set up proxy for access to pod  2) Store pod name in environment variable  3) View application output | |  | |
| **export POD\_NAME=$(**kubectl get pods -o go-template --template  ‘{{range .items}}{{.metadata.name}}  {{“\n”}}{{end}}’  **)** | |  | |
| **curl** http://localhost:8001/version | |  | |
| kubectl **exec $POD\_NAME** … | | Execute commands directly on the container once the Pod is up and running  - Parameter: name of pod | | - Any command executed after is done inside the pod  - Check that the application is up with  curl localhost:8080  - Since command is executed inside the pod, use *localhost* | |
| **Examples** | | | |
| kubectl exec $POD\_NAME **env** | | List environment variables | |
| kubectl exec -ti $POD\_NAME **bash** | | Start bash session in pod’s container | |
| kubectl proxy | | 1. Create proxy that will forward communications into the cluster-wide, private network  - Proxy  - Enables direct access to the API from terminals  - Won’t show any output while its running  - Can be terminated by pressing CTRL-C  2. See all APIs hosted through proxy endpoint/see output of application  3. Get pod name and store in environment variable  - Query pod directly through the proxy | | Example:  echo -e “\n\n\n\e[92mStarting Proxy. After starting it will not output a response. Please click the first Terminal Tab\n”;  kubectl proxy | |
| **curl** http://localhost:8001/version | | If Port 8001 is not accessible, ensure that kubectl proxy started is running | |
| export POD\_NAME=$(kubectl **get pods** -o go-template –template ‘{{range .items}}{{.metadata.name}}{{“\n}}{{end}}’)  echo Name of the Pod:  $POD\_NAME | |  | |
| kubectl **delete pod** <pod\_name> | | Delete a pod | |  | |
| kubectl **delete --all pods** | | Delete all pods | |  | |
|  | |  | |  | |
|  | |  | |  | |
| **YAML file** (pod-definition.yml) | | | | | |
| **\*\*\*** | kubectl **create -f** pod-definition.yml | | Create a pod  - Run on terminal | | # 4 basic fields  apiVersion:  kind:  metadata:  spec: |
|  | kubectl **edit pod** <pod\_name> | | Make changes to the pod definition file  - Opens up file in terminal editor automatically | |  |
|  | kubectl **apply** | | Puts into effect changes made to the pod definition file (if did not use command *edit pod*) | |  |
| **Field** |  | | | |  |
| apiVersion | - Version of API depends on what is created | | | |
| **Kind** | | **Version** | |
| POD | | v1 | |
| Service | | v1 | |
| ReplicaSet | | apps/v1 | |  |
| Deployment | | apps/v1 | |  |
| kind |  | |  | |  |
| Pod, Deployment, Config etc | | | |  |
| metadata | - In the form of a dictionary  - Examples of properties of metadata: name, labels | | | |  |
| **Example** | | | |  |
| metadata:  name: myapp-pod # string  labels: # dictionary (any key-value)  app: myapp | | | |  |
| spec | - Is a dictionary  - Provide additional information pertaining to object  - Note: different for different objects, important to refer to documentation section to get right format for each  - Can contain multiple containers (each specified with name and image) | | | |  |
| **Example** | | | |  |
| spec:  containers: # list/array  - name: nginx-container  image: nginx  - name: postgres  image: postgres  **# Pass in environment variable**  **env**:  # Array/list  - name: POSTGRES\_PASSWORD  value: mysecretpassword | | | |  |
| Example template | apiVersion: v1  kind: Pod  metadata  name: postgres  # Very helpful for ReplicaSets  labels:  tier: db-tier  spec:  containers:  - name: postgres  image: postgres | | | | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |
| **Containers** | | | | | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |
| **Code** | | **Explanation** | | | |
| **exit** | | Close container connection | |  | |
| kubectl describe pods <pod\_name> | | Under the “Containers:” section, lists the containers in the pod and their associated information | |  | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |

**Logs, Deployments, Services, ReplicaSets**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Logs** | | | | | | | | | |
| - Anything that the application would normally send to STDOUT  - Logs for container within Pod | | | | | | |  | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |
| **Code** | | | | **Explanation** | | | | | |
| kubectl **logs** | | | | Print the logs from a container in a pod | | |  | | |
| kubectl **logs $POD\_NAME** <container name> | | | | Retrieve logs  - Don’t need to specify container name if there is only one container inside the pod | | |  | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |
| **ReplicaSets** | | | | | | | | | |
| **Replication Controller**  - Controllers are processes that monitor Kubernetes objects and respond accordingly  - One controller in particular - replication controller  - Replication controller helps to run multiple instances of a single pod in the cluster at the same time (i.e. high availability)  - Even if only one pod, RC can help to bring up new pod when existing one fails  - RC creates multiple paths to share the load, can span across multiple nodes in the cluster to scale  - RC is the old technology being replaced by Replica Set | | | | | | |  | | |
| - Use pod labels as a filter for RS under the selector section  - Allows RS to know which pods to monitor  - RS will not deploy new instances of pods if number of existing pods with matching labels meet the number of replicas specified | | | | | | |  | | |
| Ways to update/scale the number of replicas:  1. Update the number of replicas in the definition file, update RS with the kubectl create… command  - Note that this does not scale the number of replicas automatically (i.e. the actual number of replicas still remains)  2. Use kubectl scale… command | | | | | | |  | | |
| When a pod containing a ReplicaSet is **deleted**,  - While pod is being terminated, another pod is being created  - ReplicaSet automatically brings up a new pod when it detects that one of the replicas is missing | | | | | | |  | | |
| When a new pod is **created**,  - ReplicaSet monitors the number of pods and makes sure that the number of pods maintains at the number that was specified  - If new pod created exceeds this specified number, it will be terminated as soon as it was created  - RS will not allow new pods to be created with the **same labels** | | | | | | |  | | |
|  | | | | | | |  | | |
|  | | | |  | | |  | | |
| **Code** | | | | **Explanation** | | | | | |
| [DEPLOYMENT-NAME]-[RANDOM-STRING] | | | | Format of name (always) | | | - RANDOM-STRING is randomly generated and uses the pod-template-hash as a seed | | |
| kubectl get rs  kubectl **get replicaset** | | | | See *ReplicaSet* created by Deployment | | |  | | |
| **Returns** | | | | | |
| DESIRED | | | **Desired state**: displays the desired number of replicas of the application (define during deployment) | | |
| CURRENT | | | Displays how many replicas are currently running | | |
| READY | | | Shows the number of pods that are ready in the RS | | |
| kubectl **scale** --replicas=[int] -f replicaset-definition.yml  **OR**  kubectl scale --replicas=[int] [TYPE] [NAME] | | | | Update the number of replicas  - This is an alternative to changing the YAML file directly (but file remains the same) | | | Example:  kubectl scale --replicas=6 replicaset myapp-replicaset | | |
| kubectl **delete** replicaset [NAME] | | | | Delete ReplicaSet and also all underlying PODS | | |  | | |
| kubectl **replace -f** [YAML file] | | | | Replace/update the RS | | |  | | |
| kubectl **create -f** replicaset-definition.yml | | | | Create ReplicaSet | | | Returns NAME, DESIRED, CURRENT, READY, AGE | | |
| kubectl **describe** replicaset | | | | See additional information about ReplicaSets | | |  | | |
| kubectl edit replicaset [RS NAME] | | | |  | | |  | | |
| # Option 1  kubectl delete replicaset [NAME]  kubectl create -f [YAML\_file]  # Option 2  kubectl edit replicaset [NAME]  kubectl delete pod [pod\_name] | | | | To fix a ReplicaSet,  - **Option 1**  - Delete and re-create the RS  - **Option 2**  - Update existing RS and then delete all PODS  - So that new pods with correct image will be created | | |  | | |
| **YAML file** (replicaset-definition.yml) | | | | | | | | | |
| **\*\*\*** | kubectl **create -f** replicaset-definition.yml | | | | | Creates replication controller | | |  |
|  | kubectl **explain replicaset** | grep VERSION | | | | | Find the correct version for the RS | | |  |
|  | **Properties** | |  | | | | | |  |
| template | | Provide a pod template to be used by the replication controller  - Essentially only the metadata and spec sections of a pod definition file (note: do proper indentation and nesting, should be children of template section) | | | | | |
| replicas | | Specify the number of replicas neeed | | | | | |
| selector | | Helps the ReplicaSet identify what parts fall under it  - RS can manage parts that were not created as part of the replica at creation  - For example, pods created before the creation of the RS which match labels specified in selector are also taken into consideration when creating replicas  - This property requires user input, has to be written in the form of *match labels* (matches pod labels to these match labels)  - Used to link RS to pods | | | | | | A major difference between a RS and RC |
| Labels under matchLabels should match with that provided in the template for the pod | | | | | |  |
|  |  | | | | |  | | |  |
|  | | | | | | | |  |
| Example template | apiVersion: **apps/v1**  kind: **ReplicaSet**  metadata:  name: myapp-replicaset  labels:  app: myapp  type: front-end  spec:  **template:**  metadata:  name: myapp-pod  labels:  app: myapp  type: front-end  spec:  containers:  - name: nginx-containers  image: nginx  **replicas:** 3  **selector:**  **matchLabels:**  type: front-end | | | | | | | | |
|  | | | |  | | |  | | |
| **Deployments** | | | | | | | | | |
| - Deployment configuration instructs Kubernetes how to create and update instances of your application  - Kubernetes master schedules application instances (to run on individual Nodes in cluster) based on Deployment  - *Kubernetes Deployment Controller*  - Continuously monitors application instances once created  - If node hosting an instance goes down/is deleted, Deployment controller replaces instance with another on another node in the cluster  - Self-healing mechanism to address machine failure or maintenance  -To specify (can be changed by updating Deployment):  - Container image for application  - Number of replicas to run  - Applications need to be packaged into one of the supported container formats in order to be deployed on Kubernetes | | | | | | | - Before there were deployments, *installation scripts* often used to start applications  - They did not allow recovery from machine failure | | |
| - Inside Kubernetes, applications run on a private, isolated network  - By default, they are visible from other pods and services within **same** Kubernetes **cluster**, but not outside that network  - i.e. applications are accessible only within cluster  - Using kubectl means interacting through an API endpoint to communicate with application  - API server will automatically create an endpoint for each application based on the application name, that is also accessible through the proxy  - Expose deployments publicly through a *Service* | | | | | | |  | | |
| Capabilities of Kubernetes deployments:  1) Requires many instances of the web server running  - When web server to be deployed in a production environment  2) Rolling updates  - Undo recent changes in the case of error  - When newer versions of application builds become available on the docker registry  - Need to upgrade docker instances seamlessly (not all at once) through rolling updates  3) Allows all the changes to be rolled out together  - When making multiple changes to environment, but don’t want to make changes immediately  - E.g. Upgrading underlying web server versions, scaling environment, modifying the resource allocations etc.  - Pause the environment, make changes and then resume | | | | | | |  | | |
| **Rollouts and Versioning**  - The creation of a deployment triggers a rollout, which in turns creates a new deployment revision  - “Application updated” means that the container version is updated to a new version  - When containers are of a new version, a new rollout is triggered to create a new deployment revision (e.g. “Revision 2”)  - This helps to keep track of deployment, enables rolling back to a previous version of deployment if necessary | | | | | | |  | | |
| Types of **deployment strategies**:  1) Recreate strategy  - Destroy all instances of old version, then deploy new instances of new version  - Problem: application down and inaccessible to users between the time that old instances are destroyed and new ones are deployed  2) Rolling Update (default)  - Take down older version and bring up a newer version one by one | | | | | | |  | | |
| Different from ReplicaSets,  - RS ensures that number of pods running always follows specified number in definition file  - Deployments are in the wider picture, they manage replica sets and the pods in them (in terms of updating versions of pods etc) | | | | | | |  | | |
|  | | | | | | |  | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |
| **Code** | | | | **Explanation** | | | | | |
| kubectl **create deployment** <deployment name> **--image=**<app image location> | | | | Deploy application/create a deployment  - Include fill repository url for images hosted outside Docker hub  - What it does:  - Search for a suitable node where an instance of the application could run  - Schedule the application to run on that pod  - Configured the cluster to reschedule the instance on a new node when needed | | | Example:  kubectl create deployment ubernetes-bootcamp –image=gcr.io/google-samples/ubernetes-bootcamp:v1 | | |
| kubectl run [name] --image=[image name] | | | | Another way of creating a deployment  - Not recommended  - Specify only the image name, not using the definition file  - Required replica sets and pods are automatically created in the backend | | |  | | |
| kubectl **get deployments** | | | | List deployments | | |  | | |
| **Returns** | | | | | |
| NAME | | | Lists names of Deployments in cluster | | |
| READY | | | Shows the ratio of CURRENT/DESIRED replicas | | |
| UP-TO-DATE | | | Displays the number of replicas that have been updated to achieve the desired state | | |
| AVAILABLE | | | Displays how many replicas/instances of the application are available to your users | | |
| AGE | | | Displays the amount of time that the application has running | | |
| **kubectl describe** <deployment type>/<name> | | | | Gives information about a deployment  - Includes the Deployment events log | | |  | | |
| kubectl describe deployment | | | | See additional information about deployments | | | Returns time deployment created, labels specified, selector, image that’s under the pod template and the status | | |
| kubectl **set image** deployment/myapp-deployment [container name]=[new image name] | | | | Update image of deployments  - Warning: results in a different configuration in the definition file | | | Example:  kubectl set image deployment/myapp-deployment nginx-container=nginx;1.12-perl  - [container name] can be found in the definition file under template/spec/containers section | | |
| kubectl **delete deployment** [deployment name] | | | | Delete deployment | | |  | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |
| **YAML file** (deployment-definition.yml) | | | | | | | | | |
| **\*\*\*** | kubectl **create -f** deployment-definition.yml | | | | Create a deployment  - **\*\*\* Automatically creates a ReplicaSet** (check with kubectl get replicaset) | | |  | |
|  | kubectl create -f deployment-definition.yml **--record** | | | | Ask K8s to record down the change cause | | | See CHANGE CAUSE with  kubectl rollout history deployment/[deployment name] | |
|  | kubectl **apply -f** deployment-definition.yml  … --record=true | | | | Update deployments  - Apply the changes made to the definition file  - Triggers new rollout and creates new revision of deployment | | |  | |
|  |  | | | |  | | |  | |
| **Field** |  | | | | | | |  | |
| matchLabels | Copy over the labels of the pod definition | | | | | | |
| Example template | apiVersion: **apps/v1**  kind: **Deployment**  metadata:  name: myapp-deployment  labels:  app: myapp  type: front-end  spec:  **template:**  metadata:  name: myapp-pod  labels:  app: myapp  type: front-end  spec:  containers:  - name: nginx-containers  image: nginx  **replicas:** 3  **selector:**  **matchLabels:**  type: front-end | | | | | | | | |
| **Services** | | | | | | | | | |
| - An abstraction which defines a logical set of Pods and a policy by which to access them  - Abstractions that allow pods to die and replicate in Kubernetes without impacting your application  - Many kinds of services available  - Discovery and routing among dependent Pods are handled by Kubernetes Services  - E.g. frontend and backend components in an application  - Allows applications running in a Kubernetes cluster to find and communicate with each other, and the outside world/traffic  - Routes traffic across a set of Pods  - Returns CNAME record with the name  - No proxy is used  - Requires v1.7 or higher of kube-dns  - Enables connectivity between groups of pods  - Without a Service, unique IP addresses of each pod are not exposed outside the cluster  - Enables external traffic exposure (for pods), load balancing and service discovery for the set of pods it defines  - Enable a loose coupling between pods  - Use cases:  - Allows communication within and outside of application (e.g. other applications, users)  - Enables frontend of application to be made available to users  - Helps communication between backend and frontend pods  - Can establish connectivity to an external data source | | | | | | |  | | |
| - Required for new deployment to be accessible without using the Proxy  - Defined using YAML (preferred) or JSON  - Targets a set of pods, usually determined by a *LabelSelector*  - *Labels and selectors*  - Core primitive grouping in Kubernetes  - Allows logical operation on objects  - Labels are key/value pairs attached to objects and can be used in any number of ways  - Designate objects for development, test and production  - Embed version tags  - Classify an object using tags  - *LabelSelector*  *-*  Labels do not provide uniqueness, expect many objects to carry the same label(s)  - Identify a set of objects using a *label selector*  - Two types of selectors: *equality-based* and *set-based*  - No logical OR (||) operator  - Ensure file statements are structured accordingly  - Can be made of multiple requirements  - For some API types like *ReplicaSets*  - Label selectors of two instances must not overlap within a namespace  - Or controller should see that as conflicting instructions, and fail to determine how many replicas should be present | | | | | | |  | | |
| - Can be exposed in different ways (specify a type)  - *ClusterIP* (default)  - Exposes the Service on an internal IP in the cluster  - Makes the Service only reachable from within the cluster  - *NodePort*  - Exposes the Service on the same port of each selected Node in the cluster using NAT  - Makes Service accessible from outside the cluster using <NodeIP>:<NodePort>  - Superset of ClusterIP  - *LoadBalancer*  - Creates an external load balancer in the current cloud (if supported)  - Assigns a fixed, external IP to the Service  - Superset of NodePort  - *ExternalName*  - Exposes the Service using an arbitrary name (specified by externalName in the spec) and no selector  - There are some uses cases with Services that involve not defining selector in the spec  - A Service created without selector will also not create corresponding Endpoints object  - Allows users to manually map a Service to specific endpoints | | | | | | |  | | |
| **NodePort** | | - Service listens to a port on the node and forward requests  - Helps makes an internal pod accessible on a pod on the node | | | | |  | | |
| **ClusterIP** | | - Service creates a virtual IP inside the cluster to enable communication between different services  - E.g. a set of frontend servers to a set of backend servers | | | | |  | | |
| **LoadBalancer** | | - Service provisions a load balancer for our application in supported cloud providers  - E.g. distribute load across to different web servers in frontend tier | | | | |  | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |
| **Code** | | | | **Explanation** | | | | | |
| kubectl **get services**  or  kubectl get svc | | | | List the current Services from cluster  - Returns NAME (service name), TYPE, CLUSTER-IP, EXTERNAL-IP, PORT(S) | | |  | | |
| **kubectl describe service** <service name>  or  kubectl describe svc <service name> | | | | Returns information about the Service | | |  | | |
| **Returns** | | | | | |
| Name, Namespace, Labels, Annotations, Selector, Type, IP, Port, TargetPort, NodePort, Endpoints, Session Affinity, External Traffic Policy, Events | | | | | |
| **kubectl expose** deployment/<pod name> --type=<”NodePort”> --name=<service name> --port <port no.> | | | | Create a new service and expose the deployment  - Service receives a unique cluster-IP, an internal port and an external-IP (IP of node) | | |  | | |
| export NODE\_PORT=$(  kubectl get **services**/<service name> -o go-template=’{{(index.spec.ports 0).nodePort}}’  )  echo NODE\_PORT=$NODE\_PORT | | | | 1. Assign the value of the Node port (port opened externally) to an environment variable  2. Test the app is exposed outside of the cluster  - Successful if there is a response from the sever | | | Reference: checking if **load-balancing** is working | | |
| curl $(minikube ip):$NODE\_PORT | | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |
|  | | | |  | | |  | | |

**NodePort, ClusterIP, Load Balancer**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **NodePort** | | | | | |
| - Listens to requests on the node  - Forwards the request on that port (on the node) to a port on the pod  - Makes an internal pod accessible through a pod on the node  - **Target port**  - Port on pod where actual application is running  - Where the service forwards the request to  - E.g. Web server is running on port 80  - **The port**  - Port on service itself  - Service is like a virtual server inside the node  - **Cluster IP of service**  - IP address of service in cluster  - **NodePort**  - Port on node itself  - Used to access application externally  - Valid range: ports 30000 to 32767 | | | | **Example:**  - Node IP: 192.168.1.1  - NodePort: 30008  - Target port/port: 80  To access application in container in pod, curl to  <http://192.168.1.1:30008>  curl <cluster-ip>:<node> | |
| - When service is created, it looks for a matching pod with the label  - If it finds more than 1 matching pod, the service automatically selects all 3 pods as endpoints to forward their request  - Random allocation to pods  - Services acts as a built-in balancer to distribute the load acros | | | |  | |
| - When deployment files is updated, service is automatically updated (highly flexible, adaptive)  - No need to change whether single pod, multiple pods, multiple nodes or not | | | |  | |
| **YAML file** (service-definition.yml) | | | | | |
| **\*\*\*** | kubectl **create -f** service-definition.yml | | Creates a service | |  |
|  | kubectl **get services** | | - View the ClusterIP, Port(s) etc | |  |
|  |  | |  | |  |
| **Field** |  | | | |  |
| **ports** | - An array  - Can have multiple port mappings within a single service | | | |
| **port** | Mandatory | | | |  |
| **targetPort** | If not provided, assumed to be the same as port | | | |  |
| **nodePort** | If not provided, a free port in the valid range is automatically allocated | | | |  |
| **selector** | - To identify exactly which pod to connect to  - Use labels of pod, refer to section **labels** in pod-definition.yml of pod | | | |  |
| Example template | apiVersion: v1  kind: Service  metadata:  name: myapp-service  spec:  type**: NodePort**  **ports:**  **- targetPort:** 80  **port:** 80  **nodePort:** 30008  **selector:**  **app: myapp**  **type: front-end** | | | | |
|  | |  | |  | |
|  | |  | |  | |
|  | |  | |  | |
| **ClusterIP** | | | | | |
| - Service creates a virtual IP inside the cluster  - Enables communication between different services  - E.g. set of front-end services to set of back-end services  - Problem 1: IP addresses of pods do not remain the same, it will not be possible to connect them to static IP addresses  - Problem 2: How to determine which  - ClusterIP helps to group the pods together  - Provides a single interface to access the pods in a group  - Requests are forwarded to one of the other pods under the service randomly  - Allows each layer (group of pods) to scale or move as required without impacting communication between the various services  - Each service is assigned a name and IP  - Name is used by other pods to access the service | | | |  | |
|  | |  | |  | |
|  | |  | |  | |
| **YAML file** (service-definition.yml) | | | | | |
| **\*\*\*** | kubectl **create -f** service-definition.yml | | Creates a service | |  |
|  | kubectl **get services** | | - View the ClusterIP, Port(s) etc  - Service can be accessed by cluster IP or service name | |  |
| **Field** |  | | | |  |
| **type** | - ClusterIP  - Default type if not specified | | | |
| **selector** | - To link service to a set of pods  - Refer to pod definition file | | | |  |
| **ports** |  | | | |  |
|  | - Create .yaml file from command line expose  kubectl expose deployment simple-webapp-deployment --name=service -- target-port=8080 --type=NodePort -- port: 8080 --dry-run-client -o yaml > svc.yaml | | | |  |
| Example template | apiVersion: v1  kind: Service  metadata:  name: back-end  spec:  type**: ClusterIP**  **ports:**  **- targetPort:** 80  **port:** 80  **selector:**  **app: myapp**  **type: back-end** | | | | |
| **Load Balancer** | | | | | |
| - Provisions a load balancer for application  - In supported cloud providers  - E.g. distribute load across different web servers in front-end tier  - If load balancer is like VirtualBox, it will be similar to NodePort  - No external load balancer configuration | | | |  | |
|  | |  | |  | |
|  | |  | |  | |
| **YAML file** (service-definition.yml) | | | | | |
| **\*\*\*** | kubectl **create -f** service-definition.yml | | Creates a service | |  |
|  | kubectl **get services** | | - View the ClusterIP, Port(s) etc  - Service can be accessed by cluster IP or service name | |  |
| **Field** |  | | | |  |
| **type** | - LoadBalancer  - Only works for supported cloud services | | | |
| Example template | apiVersion: v1  kind: Service  metadata:  name: back-end  spec:  type**: LoadBalancer**  **ports:**  **- targetPort:** 80  **port:** 80  **selector:**  **app: myapp**  **type: back-end**  **nodePort: 30008** | | | | |

**Scaling, Rolling Update, Load Balancing**

|  |  |  |
| --- | --- | --- |
| **Scaling** | | |
| - Need to scale application to keep up with user demand when traffic increases  - Scale by changing the number of replicas  - A deployment can also be started with multiple instances  - Ensure that new pods are created and scheduled to Nodes with available resources  - *Autoscaling* of Pods is supported by Kubernetes  - Scaling to zero is also possible, and it will terminate all Pods of the specified Deployment  - Since we are running multiple instances of an application, Services:  - Have an integrated load-balancer to distributed network traffic to all Pods of an exposed Deployment  - Monitor continuously the running Pods using endpoints, ensure that traffic is sent only to available Pods  - With multiple instances of an Application running, Rolling updates can be done without downtime | |  |
| - Scaling increases the number of pods, each with different IP addresses  - Change in scaling is registered in the Deployments events log  - Scaling up means creating a new pod with a new instance of the same application (don’t add containers to an existing pod)  - When current node has no sufficient capacity, new pods can be deployed on a new node in the cluster | |  |
|  |  |  |
|  |  |  |
| **Code** | **Explanation** | |
| kubectl **scale** <deployment type>/<name> --replicas=<desired no. of instances> | Scale up or down the Service and number of instances of the application |  |
| kubectl **get deployments** | List the number of replicas |  |
| kubectl **get pods -o wide** | List the number of pods | Returns NAME, READY, STATUS, RESTARTS, AGE, IP, NODE  - NODE tells which pods are running on the same node |
|  |  |  |
| **Rolling Update & Rollbacks** | | |
| - Allow Deployments’ update to take place with zero downtime by **incrementally updating Pods instances with new ones**  - New Pods will be scheduled on Nodes with available resources  - Perform updates without affecting application availability  - By default  - Maximum number of Pods that unavailable during update is 1  - Maximum number of new Pods that can be created is 1  - Can be configured to either numbers or percentages of Pods  - In Kubernetes, updates are versioned (like git) and any Deployment update can be reverted to a previous (stable) version | |  |
| - Similar to scaling, if a Deployment is exposed publicly, the Service will load-balance the traffic only to *available Pods* during the update  - *Available pods*  - Instances that are available to users of the application  - Rolling updates allow the following actions:  - Promote an application from one environment to another (via container image updates)  - Rollback to previous versions  - Continuous Integration and Continuous Delivery of applications with zero downtime | |  |
| **Upgrades/Downgrades** (what happens under-the-hood)  - Kubernetes deployment object creates a new replica set under-the-hood  - At the same time, taking down pods in the old replica set  - In this sense, a new replica set is created  - The old replica set will have 0 pods  **Rollbacks**  - Roll back to a previous version and undo changes to instances  - Under-the-hood, deployment will destroy the pods in the new replica set (updated pods with changes applied), and bring up older ones in the old replica set  - Both replica sets continue to remain, just that the number of pods in them are reversed | |  |
| Scenario: Upgrade is to an image does not exist  - Considered an upgrade  Example:  - Kubernetes will try to terminate one instance from the old working version (brings total no. of running pods to 5)  - Also, tries to create 3 new pods of the new version (but is not successful - STATUS is ImagePullBackOff which means that the deployment is unable to pull the image from docker hub)  - Since unable to deploy any of the 3 new pods, Kubernetes proactively stops proceeding with the new upgrade and also stops terminating the old replica set  - It knows that terminating all old versions and deploying all new versioned pods will impact the users since image does not exist  - **Proactively stops new upgrade, starts to wait for new image to be available**  - Undo this by doing a rollback | | Example return from kubectl get deployment  NAME myapp-deployment  DESIRED 6  CURRENT 8  UP-TO-DATE 3  AVAILABLE 5  AGE 11m |
|  | |  |
|  |  |  |
|  |  |  |
|  |  |  |
| **Code** | **Explanation** | |
| kubectl **set image** <deployment type>/<name> <container name>=<image name> … | Update the image of the application  - Command notifies the Deployment to use a different image for the app and initiate a rolling update | For example, update the image of the application to version 2  kubectl **set image** deployment/nginx busybox=busybox nginx=nginx:1.9.1 |
| kubectl rollout status <deployment type>/<name>  kubectl **rollout status deployment/<**deployment name> | Check status of rollout, whether successful or not |  |
| kubectl **rollout history** deployment/<deployment name> | Shows the revisions and history of the deployment | Returns REVISION, CHANGE-CAUSE |
| kubectl **rollout undo** <deployment type>/<name> | Roll back to previously working version  - Reverts deployment to previous known state (v2 of the image) | For example,  kubectl rollout undo deployment/myapp-deployment  - In rollout history, there will be missing revision numbers in the list since the rollback reapplies a previous command (which is pulled out of the list and added to the back as the latest revision) |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| **Load Balancing** | | |
| - To demonstrate that load-balancing (of the traffic) is working  - Refer to code in Services on how to do steps 1-3  1) Find exposed IP and Port using kubectl describe services  2) Create an environment variable called NODE\_PORT as value of Node port  3) Do a curl to exposed IP and post **multiple times**  4) If load-balancing is working, a different Pod is hit with every request | |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| **Code** | **Explanation** | |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| **Networking** | | |
| **Single-node Cluster**  - In Docker, IP address is always assigned to container  - In Kubernetes, IP address is always assigned to **pod** - each pod gets its own **internal** **IP address**  - Each node has its own internal IP address network to which the pods are connected to  - When Kubernetes initially configured, an internal private network is created and all pods are attached to it (each assigned a separate IP from this network)  - Pods can communicate within this internal IP but not good to use it to connect to other parts since IP addresses are subject to change when pods are recreated | |  |
| **Cluster Networking**  - Cluster with multiple nodes  - Need to take note and ensure:  - All containers/PODs can communicate to one another without configuring NAT  - All nodes can communicate with all containers and vice-versa without NAT  - Multiple pre-built solutions available to help with meeting the requirements above:  - CISCO, cilium, flannel, vmwareNSX  - Depends on the platform on which Kubernetes cluster is deployed on  - Manages networks and IPs in nodes and assigns a different network address for each network in the node  - Creates a virtual network of all pods and node where they are all assigned a unique IP address  - Using simple routing techniques, cluster networking enables communication between the different pods/nodes to meet the networking requirements of Kubernetes  - Meeting requirements, all pods can communicate with each other using the assigned IP address | |  |
|  | |  |
|  | |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  | |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| **Microservices Architecture** | | |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  |  |  |
|  |  | |
| **docker run -d --name=vote -p 5000:80 --link redis:redis voting-app** | Link containers together   * Creates an entry in the etc/hosts file of container with internal IP of connecting container |  |
| **kubectl service <service\_name> --url** | Find the url to access a pod |  |
| kubectl get pods**,**svc,deployments | See *both* pods, services, deployments available in K8s cluster   * Use comma |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**Setting Up (Virtual Environment, Docker, K8s)**

|  |  |  |
| --- | --- | --- |
| **VirtualBox** | | |
| 1. Go to <https://www.virtualbox.org/>  - Download latest version for Windows  2. Go to <https://www.osboxes.org/>  - Download a VB image (.vdi) like Ubuntu  3. Create a new machine using the image in VB  - Change settings > Network where adapter to attach to is “Bridged Adapter”  - Will bridge to the WiFi router  4. Power up virtual machine  5. Check IP address assigned to VM by wireless router  - Check on terminal with **ifconfig** (Ubuntu)**/ipconfig** (Windows)  - Look under “inet addr”/“IPv4 Address”  6. Use a SSH terminal to SSH into the VM  - For example, MobaXterm or PuTTY  - If error in connected, check if SSH service has been set up - run on the Ubuntu terminal, service ssh status  7. Set up SSH service  - Run on terminal  apt-get update, apt-get install openssh-server  8. Restart session in SSH terminal  9. Enter password to log in to VM through SSH client | |  |
| **Docker** | | |
| Run on VM terminal, **apt-get install docker.io** | |  |
|  |  |  |
|  |  |  |
|  |  |  |
| **Code** | **Explanation** | |
| docker | Shows help page |  |
| docker **version** | Shows version of Docker in use |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| **MiniKube** | | |
| - Lightweight Kubernetes implementation  - Creates a VM on your local machine  - Deploys a simple cluster container only one node  - Minikube CLI provides basic bootstrapping operations for working with your cluster (includes *start, stop, status, delete* etc) | |  |
| - Bundles all different components found in Master and Worker node into a single image provided as a single K8s cluster  - Available and bundled as an image and available online for download  - Things needed:  - Hypervisor  - kubectl utility  - MiniKube executable installed on system | |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| **Code** | **Explanation** | |
| minikube start --vm-driver=<driver\_name> | Install MiniKube (run on terminal) |  |
| minikube version |  |  |
| minikube start | Start a cluster |  |
| kubectl run hello-minikube | Deploy an application on the cluster |  |
| kubectl cluster-info | View information about the cluster |  |
| kubectl get nodes | List all nodes part of the cluster |  |
| kubectl run my-web-app --image=my-web-app --replicas=100 | Run many instances of applications across many nodes |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **YAML** | | | | | | |
| - Used to represent data  - In Kubernetes, YAML used as inputs for creation of objects (e.g. PODs, replicas, deployment services) | | | | | |  |
| Key characteristics | | YAML example | | |  |  |
| Comment | | # this is a comment | | |  |
| Key-value pair | | Fruit: Apple  Meat: Chicken  Vegetable: Carrot | | |  |
| Array/lists  (ordered collection) | | Fruits:  - Orange  - Apple  - Banana | | | - Each dash indicates an element of the array |
| Dictionary/Map  (Unordered collection) | | Banana:  Calories: 105  Fat: 0.4 g  Carbs: 27 g | | | - There must be **equal number of blank spaces** before the properties of a single item (e.g. item here is banana)  - Or else, properties will fall under the first item that is indent above it |
| List of dictionaries | | Fruits:  - Banana:  Calories: 105  Fat: 0.4 g  Carbs: 27 g  - Grape:  …  # OR  - Month: June  Wage: 3000  - Month: July  Wage: 3500 | | |  |
|  | | |  | | |  |
|  | | |  | | |  |
|  | | |  | | |  |
| **Basic Kubernetes definition file** | | | | | |  |
| - Always contains 4 top level fields  - Required fields that **must** be present in the configuration file  - The specification (spec) section is the most important as it defines what Kubernetes object is created | | | | | |  |
| # Version of K8s API used to create objects  **apiVersion: …**  # Type of objected created  **kind: …**  # Data about object (name, labels etc)  **metadata: …**  # Specification section  **spec: …** | | | | | |  |
|  |  | | |  | |  |
|  |  | | |  | |  |
|  |  | | |  | |  |
|  | | |  | | |  |
|  | | |  | | |  |
|  | | |  | | |  |

**Contents**

**1. Cluster, Master, Nodes, Pods, Containers**

**2. Scaling, Rolling Update, Load Balancing**

**3. MiniKube**

**4. Setting Up Environment in Virtual Machine**

**5. YAML**