Version v1.13

[Minikube](https://kubernetes.io/docs/setup/minikube/) is a method for creating a local, single-node Kubernetes cluster for development and testing. Setup is completely automated and doesn’t require a cloud provider account.

[Kubeadm-dind](https://github.com/kubernetes-sigs/kubeadm-dind-cluster) is a multi-node (while minikube is single-node) Kubernetes cluster which only requires a docker daemon. It uses docker-in-docker technique to spawn the Kubernetes cluster.

[Amazon Elastic Container Service for Kubernetes](https://aws.amazon.com/eks/) offers managed Kubernetes service.

[Kubermatic](https://www.loodse.com/) provides managed Kubernetes clusters for various public clouds, including AWS and Digital Ocean, as well as on-premises with OpenStack integration.

Master node(s)

| Protocol | Direction | Port Range | Purpose | Used By |
| --- | --- | --- | --- | --- |
| TCP | Inbound | 6443\* | Kubernetes API server | All |
| TCP | Inbound | 2379-2380 | etcd server client API | kube-apiserver, etcd |
| TCP | Inbound | 10250 | Kubelet API | Self, Control plane |
| TCP | Inbound | 10251 | kube-scheduler | Self |
| TCP | Inbound | 10252 | kube-controller-manager | Self |

Worker node(s)

| Protocol | Direction | Port Range | Purpose | Used By |
| --- | --- | --- | --- | --- |
| TCP | Inbound | 10250 | Kubelet API | Self, Control plane |
| TCP | Inbound | 30000-32767 | NodePort Services\*\* | All |

* **kubeadm**: the command to bootstrap the cluster.
* **kubelet**: the component that runs on all of the machines in your cluster and does things like starting pods and containers.
* **kubectl**: the command line util to talk to your cluster.

**kubeadm init**

You can install a pod network add-on with the following command:

**kubectl apply -f <add-on.yaml>**

**kubectl get pods --all-namespaces**.

By default, your cluster will not schedule pods on the master for security reasons. If you want to be able to schedule pods on the master,

**kubectl taint nodes --all node-role.kubernetes.io/master-**

Joining your nodes

**kubeadm join --token <token> <master-ip>:<master-port> --discovery-token-ca-cert-hash sha256:<hash>**

If you do not have the token, you can get it by running the following command on the master node:

**kubeadm token list**

By default, tokens expire after 24 hours. If you are joining a node to the cluster after the current token has expired, you can create a new token by running the following command on the master node:

**kubeadm token create**

If you want to connect to the API Server from outside the cluster you can use **kubectl proxy**

[drain the node](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#drain) and make sure that the node is empty before shutting it down.

**kubectl drain <node name> --delete-local-data --force --ignore-daemonsets**

**kubectl delete node <node name>**

The components are defined using the following fields:

* **apiServer**
* **controllerManager**
* **scheduler**

APIServer flags

**apiVersion: kubeadm.k8s.io/v1beta1**

**kind: ClusterConfiguration**

**kubernetesVersion: v1.13.0**

**metadata:**

**name: 1.13-sample**

**apiServer:**

**extraArgs:**

**advertise-address: 192.168.0.103**

**anonymous-auth: false**

**enable-admission-plugins: AlwaysPullImages,DefaultStorageClass**

**audit-log-path: /home/johndoe/audit.log**

ControllerManager flags

**apiVersion: kubeadm.k8s.io/v1beta1**

**kind: ClusterConfiguration**

**kubernetesVersion: v1.13.0**

**metadata:**

**name: 1.13-sample**

**controllerManager:**

**extraArgs:**

**cluster-signing-key-file: /home/johndoe/keys/ca.key**

**bind-address: 0.0.0.0**

**deployment-controller-sync-period: 50**

Scheduler flags

**apiVersion: kubeadm.k8s.io/v1beta1**

**kind: ClusterConfiguration**

**kubernetesVersion: v1.13.0**

**metadata:**

**name: 1.13-sample**

**scheduler:**

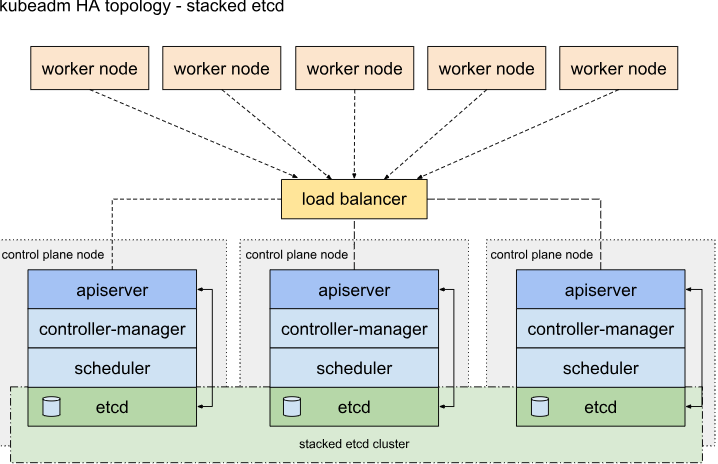
**extraArgs:**

**address: 0.0.0.0**

**config: /home/johndoe/schedconfig.yaml**

**kubeconfig: /home/johndoe/kubeconfig.yaml**

The **kube-apiserver** is exposed to worker nodes using a load balance



Type the following and watch the pods of the components get started:

**kubectl get pod -n kube-system -w**

Kubernetes Basics Modules

1. create a k8’s cluster
2. Deploy an application
3. Explore your application
4. Expose your app publicly
5. Scale up u r app
6. Update u r app

**Kubernetes coordinates a highly available cluster of computers that are connected to work as a single unit.**  **Kubernetes automates the distribution and scheduling of application containers across a cluster in a more efficient way.**

**1.create a k8’s cluster**

A Kubernetes cluster consists of two types of resources:

* The **Master** coordinates the cluster
* **Nodes** are the workers that run applications

The master coordinates all activities in your cluster, such as scheduling applications, maintaining applications' desired state, scaling applications, and rolling out new updates.

**A node is a VM or a physical computer that serves as a worker machine in a Kubernetes cluster.** Each node has a Kubelet, which is an agent for managing the node and communicating with the Kubernetes master. A Kubernetes cluster that handles production traffic should have a minimum of three nodes.

**The nodes communicate with the master using the Kubernetes API**, which the master exposes. End users can also use the Kubernetes API directly to interact with the cluster.

1. Minikube version
2. Minikube start
3. Kubectl version
4. Kubectl cluster info
5. kubectl get nodes

**2. Deploy an application**

Using kubectl to Create a Deployment

Once you have a running Kubernetes cluster, you can deploy your containerized applications on top of it. To do so, you create a Kubernetes **Deployment** configuration

Once you've created a Deployment, the Kubernetes master schedules mentioned application instances onto individual Nodes in the cluster.

Once the application instances are created, a Kubernetes Deployment Controller continuously monitors those instances. If the Node hosting an instance goes down or is deleted, the Deployment controller replaces it. **This provides a self-healing mechanism to address machine failure or maintenance.**

You can create and manage a Deployment by using the Kubernetes command line interface, **Kubectl**. Kubectl uses the Kubernetes API to interact with the cluster.

kubectl version

kubectl get nodes

Let’s run our first app on Kubernetes with the kubectl run command. The run command creates a new deployment.

kubectl run kubernetes-bootcamp --image=gcr.io/google-samples/kubernetes-bootcamp:v1 --port=8080

We need to provide the deployment name and app image location (include the full repository url for images hosted outside Docker hub). We want to run the app on a specific port so we add the --portparameter:

* scheduled the application to run on that Node
* configured the cluster to reschedule the instance on a new Node when needed
* To list your deployments use the get deployments command:
* kubectl get deployments

Pods that are running inside Kubernetes are running on a private, isolated network. By default they are visible from other pods and services within the same kubernetes cluster, but not outside that network. When we use kubectl, we're interacting through an API endpoint to communicate with our application

The kubectl command can create a proxy that will forward communications into the cluster-wide, private network. The proxy can be terminated by pressing control-C and won't show any output while its running.

You can see all those APIs hosted through the proxy endpoint, now available at through [http://localhost:8001](http://localhost:8001/). For example, we can query the version directly through the API using the curl command:

curl http://localhost:8001/version

The API server will automatically create an endpoint for each pod, based on the pod name, that is also accessible through the proxy.

export POD\_NAME=$(kubectl get pods -o go-template --template '{{range .items}}{{.metadata.name}}{{"\n"}}{{end}}') echo Name of the Pod: $POD\_NAME

Now we can make an HTTP request to the application running in that pod:

curl <http://localhost:8001/api/v1/namespaces/default/pods/$POD_NAME/proxy/>

**3.Explore your application**

Viewing Pods and Nodes

When you created a Deployment in Module [2](https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-intro/), Kubernetes created a **Pod** to host your application instance. A Pod is a Kubernetes abstraction that represents a group of one or more application containers (such as Docker or rkt), and some shared resources for those containers. Those resources include:

* Shared storage, as Volumes
* Networking, as a unique cluster IP address
* Information about how to run each container, such as the container image version or specific ports to use

The containers in a Pod share an IP Address and port space, are always co-located and co-scheduled, and run in a shared context on the same Node.

Pods are the atomic unit on the Kubernetes platform. When we create a Deployment on Kubernetes, that Deployment creates Pods with containers inside them (as opposed to creating containers directly). Each Pod is tied to the Node where it is scheduled, and remains there until termination (according to restart policy) or deletion. In case of a Node failure, identical Pods are scheduled on other available Nodes in the cluster.

*A Pod is a group of one or more application containers (such as Docker or rkt) and includes shared storage (volumes), IP address and information about how to run them.*

A Pod always runs on a **Node**. A Node is a worker machine in Kubernetes and may be either a virtual or a physical machine, depending on the cluster.  Each Node is managed by the Master. A Node can have multiple pods, and the Kubernetes master automatically handles scheduling the pods across the Nodes in the cluster.

Every Kubernetes Node runs at least:

* Kubelet, a process responsible for communication between the Kubernetes Master and the Node; it manages the Pods and the containers running on a machine.
* A container runtime (like Docker, rkt) responsible for pulling the container image from a registry, unpacking the container, and running the application.

*Containers should only be scheduled together in a single Pod if they are tightly coupled and need to share resources such as disk.*

* **kubectl get** - list resources
* **kubectl describe** - show detailed information about a resource
* **kubectl logs** - print the logs from a container in a pod
* **kubectl exec** - execute a command on a container in a pod
* kubectl get command and look for existing Pods:
* kubectl get pods
* Next, to view what containers are inside that Pod and what images are used to build those containers we run the describe podscommand:
* kubectl describe pods
* We can retrieve these logs using the kubectl logs command:
* kubectl logs $POD\_NAME
* We can execute commands directly on the container once the Pod is up and running. For this, we use the exec command and use the name of the Pod as a parameter. Let’s list the environment variables:
* kubectl exec $POD\_NAME env
* Next let’s start a bash session in the Pod’s container:
* kubectl exec -ti $POD\_NAME bash

To close your container connection type exit

**4. Using a service to expose your app**

Kubernetes [Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) are mortal. Pods in fact have a [lifecycle](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/). When a worker node dies, the Pods running on the Node are also lost.  A [ReplicaSet](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/) might then dynamically drive the cluster back to desired state via creation of new Pods to keep your application running.

the front-end system should not care about backend replicas or even if a Pod is lost and recreated. That said, each Pod in a Kubernetes cluster has a unique IP address, even Pods on the same Node, so there needs to be a way of automatically reconciling changes among Pods so that your applications continue to function.

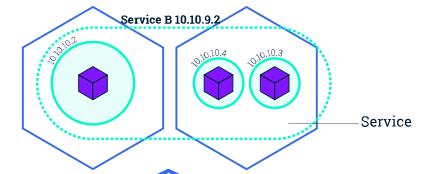
A Service in Kubernetes is an abstraction which defines a logical set of Pods and a policy by which to access them. Services enable a loose coupling between dependent Pods. A Service is defined using YAML [(preferred)](https://kubernetes.io/docs/concepts/configuration/overview/#general-config-tips) or JSON, like all Kubernetes objects. The set of Pods targeted by a Service is usually determined by a *Label Selector* (see below for why you might want a Service without including **selector** in the spec).

Although each Pod has a unique IP address, those IPs are not exposed outside the cluster without a Service. Services allow your applications to receive traffic. Services can be exposed in different ways by specifying a **type** in the ServiceSpec:

* *ClusterIP* (default) - Exposes the Service on an internal IP in the cluster. This type 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 a Service accessible from outside the cluster using **<NodeIP>:<NodePort>**. Superset of ClusterIP.
* *LoadBalancer* - Creates an external load balancer in the current cloud (if supported) and 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) by returning a CNAME record with the name. No proxy is used. This type requires v1.7 or higher of **kube-dns**.

A Service created without **selector** will also not create the corresponding Endpoints object.

This allows users to manually map a Service to specific endpoints. Another possibility why there may be no selector is you are strictly using **type: ExternalName**.



A Service routes traffic across a set of Pods. Discovery and routing among dependent Pods (such as the frontend and backend components in an application) is handled by Kubernetes Services.

Services match a set of Pods using [labels and selectors](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels),

 Labels are key/value pairs attached to objects and can be used in any number of ways:

We’ll use the kubectl get command and look for existing Pods:

kubectl get pods

Next let’s list the current Services from our cluster:

kubectl get services

To create a new service and expose it to external traffic we’ll use the expose command with NodePort as parameter (minikube does not support the LoadBalancer option yet).

kubectl expose deployment/kubernetes-bootcamp --type="NodePort" --port 8080

kubectl get services

To find out what port was opened externally (by the NodePort option) we’ll run the describe service command:

kubectl describe services/kubernetes-bootcamp

export NODE\_PORT=$(kubectl get services/kubernetes-bootcamp -o go-template='{{(index .spec.ports 0).nodePort}}') echo NODE\_PORT=$NODE\_PORT

kubectl describe deployment

kubectl get pods -l run=kubernetes-bootcamp

kubectl get services -l run=kubernetes-bootcamp

To apply a new label we use the label command followed by the object type, object name and the new label:

kubectl label pod $POD\_NAME app=v1

This will apply a new label to our Pod (we pinned the application version to the Pod), and we can check it with the describe pod command:

kubectl describe pods $POD\_NAME

We see here that the label is attached now to our Pod. And we can query now the list of pods using the new label:

kubectl get pods -l app=v1

To delete Services you can use the delete service command. Labels can be used also here:

kubectl delete service -l run=kubernetes-bootcamp

Confirm that the service is gone:

kubectl get services

kubectl exec -ti $POD\_NAME curl localhost:8080

1. **Scale your App**

The Deployment created only one Pod for running our application. When traffic increases, we will need to scale the application to keep up with user demand.

**Scaling** is accomplished by changing the number of replicas in a Deployment

Scaling to zero is also possible, and it will terminate all Pods of the specified Deployment.

Running multiple instances of an application will require a way to distribute the traffic to all of them. Services have an integrated load-balancer that will distribute network traffic to all Pods of an exposed Deployment. Services will monitor continuously the running Pods using endpoints, to ensure the traffic is sent only to available Pods.

*Scaling is accomplished by changing the number of replicas in a Deployment.*

Once you have multiple instances of an Application running, you would be able to do Rolling updates without downtime.

The DESIRED state is showing the configured number of replicas

The CURRENT state show how many replicas are running now

The UP-TO-DATE is the number of replicas that were updated to match the desired (configured) state

The AVAILABLE state shows how many replicas are actually AVAILABLE to the users

Next, let’s scale the Deployment to 4 replicas. We’ll use the kubectl scale command, followed by the deployment type, name and desired number of instances:

kubectl scale deployments/kubernetes-bootcamp --replicas=4

The change was applied, and we have 4 instances of the application available. Next, let’s check if the number of Pods changed:

kubectl get pods -o wide

There are 4 Pods now, with different IP addresses. The change was registered in the Deployment events log. To check that, use the describe command:

k To scale down the Service to 2 replicas, run again the scale command:

kubectl scale deployments/kubernetes-bootcamp --replicas=2

ubectl describe deployments/kubernetes-bootcamp

**7.Update your App(RollingUpdate)**

Users expect applications to be available all the time and developers are expected to deploy new versions of them several times a day.  In Kubernetes this is done with rolling updates. **Rolling updates** allow Deployments' update to take place with zero downtime by incrementally updating Pods instances with new ones. The new Pods will be scheduled on Nodes with available resources.

In the previous module we scaled our application to run multiple instances. This is a requirement for performing updates without affecting application availability. By default, the maximum number of Pods that can be unavailable during the update and the maximum number of new Pods that can be created, is one. Both options can be configured to either numbers or percentages (of Pods). In Kubernetes, updates are versioned and any Deployment update can be reverted to previous (stable) version.

Summary:

* Updating an app

*Rolling updates allow Deployments' update to take place with zero downtime by incrementally updating Pods instances with new ones.*

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

Similar to application Scaling, if a Deployment is exposed publicly, the Service will load-balance the traffic only to available Pods during the update. An available Pod is an instance that is available to the users of the application.

To list your deployments use the get deployments command: kubectl get deployments

To list the running Pods use the get podscommand:

kubectl get pods

To view the current image version of the app, run a describe command against the Pods (look at the Image field):

kubectl describe pods

To update the image of the application to version 2, use the set image command, followed by the deployment name and the new image version:

kubectl set image deployments/kubernetes-bootcamp kubernetes-bootcamp=jocatalin/kubernetes-bootcamp:v2

The command notified the Deployment to use a different image for your app and initiated a rolling update. Check the status of the new Pods, and view the old one terminating with the get pods command:

kubectl get pods

First, let’s check that the App is running. To find out the exposed IP and Port we can use describe service:

kubectl describe services/kubernetes-bootcamp

Create an environment variable called NODE\_PORT that has the value of the Node port assigned:

export NODE\_PORT=$(kubectl get services/kubernetes-bootcamp -o go-template='{{(index .spec.ports 0).nodePort}}') echo NODE\_PORT=$NODE\_PORT

The update can be confirmed also by running a rollout status command:

kubectl rollout status deployments/kubernetes-bootcamp

To view the current image version of the app, run a describe command against the Pods:

kubectl describe pods

We run now version 2 of the app (look at the Image field)

Let’s perform another update, and deploy image tagged as v10 :

kubectl set image deployments/kubernetes-bootcamp kubernetes-bootcamp=gcr.io/google-samples/kubernetes-bootcamp:v10

Use get deployments to see the status of the deployment:

kubectl get deployments

And something is wrong… We do not have the desired number of Pods available. List the Pods again:

kubectl get pods

A describe command on the Pods should give more insights:

kubectl describe pods

There is no image called v10 in the repository. Let’s roll back to our previously working version. We’ll use the rollout undo command:

kubectl rollout undo deployments/kubernetes-bootcamp

The rollout command reverted the deployment to the previous known state (v2 of the image). Updates are versioned and you can revert to any previously know state of a Deployment. List again the Pods:

kubectl get pods

Four Pods are running. Check again the image deployed on the them:

kubectl describe pods

We see that the deployment is using a stable version of the app (v2). The Rollback was successful.

first create a ConfigMap from the **redis-config** file:

pods/config/redis-config

**curl -OL https://k8s.io/examples/pods/config/redis-config**

**kubectl create configmap example-redis-config --from-file=redis-config**

**kubectl get configmap example-redis-config -o yaml**

**apiVersion: v1**

**data:**

**redis-config: *|***

***maxmemory 2mb***

***maxmemory-policy allkeys-lru***

**kind: ConfigMap**

**metadata:**

**creationTimestamp: 2016-03-30T18:14:41Z**

**name: example-redis-config**

**namespace: default**

**resourceVersion: "24686"**

**selfLink: /api/v1/namespaces/default/configmaps/example-redis-config**

**uid: 460a2b6e-f6a3-11e5-8ae5-42010af00002**

**apiVersion: v1**

**kind: Pod**

**metadata:**

**name: redis**

**spec:**

**containers:**

**- name: redis**

**image: kubernetes/redis:v1**

**env:**

**- name: MASTER**

**value: "true"**

**ports:**

**- containerPort: 6379**

**resources:**

**limits:**

**cpu: "0.1"**

**volumeMounts:**

**- mountPath: /redis-master-data**

**name: data**

**- mountPath: /redis-master**

**name: config**

**volumes:**

**- name: data**

**emptyDir: {}**

**- name: config**

**configMap:**

**name: example-redis-config**

**items:**

**- key: redis-config**

**path: redis.conf**

Use **kubectl exec** to enter the pod and run the **redis-cli** tool to verify that the configuration was correctly applied:

**kubectl exec -it redis redis-cli**

Create the pod:

**kubectl create -f https://k8s.io/examples/pods/config/redis-pod.yaml**

Configure **kubectl** to communicate with your Kubernetes API server.

* 1. Run a Hello World application in your cluster:

**kubectl run hello-world --replicas=5 --labels="run=load-balancer-example" --image=gcr.io/google-samples/node-hello:1.0 --port=8080**

Display information about the Deployment:

**kubectl get deployments hello-world**

**kubectl describe deployments hello-world**

Display information about your ReplicaSet objects:

**kubectl get replicasets**

**kubectl describe replicasets**

Create a Service object that exposes the deployment:

**kubectl expose deployment hello-world --type=LoadBalancer --name=my-service**

Display information about the Service:

**kubectl get services my-service**

Display detailed information about the Service:

**kubectl describe services my-service**

To verify these are pod addresses, enter this command:

**kubectl get pods --output=wide**

To delete the Service, enter this command:

**kubectl delete services my-service**

To delete the Deployment, the ReplicaSet, and the Pods that are running the Hello World application, enter this command:

**kubectl delete deployment hello-world**

### Creating the Redis Master Deployment

The manifest file, included below, specifies a Deployment controller that runs a single replica Redis master Pod.

**apiVersion: apps/v1 *# for versions before 1.9.0 use apps/v1beta2***

**kind: Deployment**

**metadata:**

**name: redis-master**

**labels:**

**app: redis**

**spec:**

**selector:**

**matchLabels:**

**app: redis**

**role: master**

**tier: backend**

**replicas: 1**

**template:**

**metadata:**

**labels:**

**app: redis**

**role: master**

**tier: backend**

**spec:**

**containers:**

**- name: master**

**image: k8s.gcr.io/redis:e2e *# or just image: redis***

**resources:**

**requests:**

**cpu: 100m**

**memory: 100Mi**

**ports:**

**- containerPort: 6379**

Apply the Redis Master Deployment from the **redis-master-deployment.yaml** file:

**kubectl apply -f https://k8s.io/examples/application/guestbook/redis-master-deployment.yaml**

Query the list of Pods to verify that the Redis Master Pod is running:

**kubectl get pods**

Run the following command to view the logs from the Redis Master Pod:

**kubectl logs -f POD-NAME**

### Creating the Redis Master Service

The guestbook applications needs to communicate to the Redis master to write its data. You need to apply a [Service](https://kubernetes.io/docs/concepts/services-networking/service/) to proxy the traffic to the Redis master Pod. A Service defines a policy to access the Pod

**apiVersion: v1**

**kind: Service**

**metadata:**

**name: redis-master**

**labels:**

**app: redis**

**role: master**

**tier: backend**

**spec:**

**ports:**

**- port: 6379**

**targetPort: 6379**

**selector:**

**app: redis**

**role: master**

**tier: backend**

Apply the Redis Master Service from the following **redis-master-service.yaml** file:

**kubectl apply -f https://k8s.io/examples/application/guestbook/redis-master-service.yaml**

Query the list of Services to verify that the Redis Master Service is running:

**kubectl get service**

**apiVersion: apps/v1 *# for versions before 1.9.0 use apps/v1beta2***

**kind: Deployment**

**metadata:**

**name: redis-slave**

**labels:**

**app: redis**

**spec:**

**selector:**

**matchLabels:**

**app: redis**

**role: slave**

**tier: backend**

**replicas: 2**

**template:**

**metadata:**

**labels:**

**app: redis**

**role: slave**

**tier: backend**

**spec:**

**containers:**

**- name: slave**

**image: gcr.io/google\_samples/gb-redisslave:v1**

**resources:**

**requests:**

**cpu: 100m**

**memory: 100Mi**

**env:**

**- name: GET\_HOSTS\_FROM**

**value: dns**

***# Using `GET\_HOSTS\_FROM=dns` requires your cluster to***

***# provide a dns service. As of Kubernetes 1.3, DNS is a built-in***

***# service launched automatically. However, if the cluster you are using***

***# does not have a built-in DNS service, you can instead***

***# access an environment variable to find the master***

***# service's host. To do so, comment out the 'value: dns' line above, and***

***# uncomment the line below:***

***# value: env***

**ports:**

**- containerPort: 6379**

Query the list of Pods to verify that the Redis Slave Pods are running:

**kubectl get pods**

### Creating the Redis Slave Service

The guestbook application needs to communicate to Redis slaves to read data. To make the Redis slaves discoverable, you need to set up a Service. A Service provides transparent load balancing to a set of Pods.

**apiVersion: v1**

**kind: Service**

**metadata:**

**name: redis-slave**

**labels:**

**app: redis**

**role: slave**

**tier: backend**

**spec:**

**ports:**

**- port: 6379**

**selector:**

**app: redis**

**role: slave**

**tier: backend**

**kubectl apply -f https://k8s.io/examples/application/guestbook/redis-slave-service.yaml**

**kubectl get services**

 It is configured to connect to the **redis-master** Service for write requests and the **redis-slave** service for Read requests.

**apiVersion: apps/v1 *# for versions before 1.9.0 use apps/v1beta2***

**kind: Deployment**

**metadata:**

**name: frontend**

**labels:**

**app: guestbook**

**spec:**

**selector:**

**matchLabels:**

**app: guestbook**

**tier: frontend**

**replicas: 3**

**template:**

**metadata:**

**labels:**

**app: guestbook**

**tier: frontend**

**spec:**

**containers:**

**- name: php-redis**

**image: gcr.io/google-samples/gb-frontend:v4**

**resources:**

**requests:**

**cpu: 100m**

**memory: 100Mi**

**env:**

**- name: GET\_HOSTS\_FROM**

**value: dns**

***# Using `GET\_HOSTS\_FROM=dns` requires your cluster to***

***# provide a dns service. As of Kubernetes 1.3, DNS is a built-in***

***# service launched automatically. However, if the cluster you are using***

***# does not have a built-in DNS service, you can instead***

***# access an environment variable to find the master***

***# service's host. To do so, comment out the 'value: dns' line above, and***

***# uncomment the line below:***

***# value: env***

**ports:**

**- containerPort: 80**

**kubectl apply -f https://k8s.io/examples/application/guestbook/frontend-deployment.yaml**

Query the list of Pods to verify that the three frontend replicas are running:

**kubectl get pods -l app=guestbook -l tier=frontend**

**apiVersion: v1**

**kind: Service**

**metadata:**

**name: frontend**

**labels:**

**app: guestbook**

**tier: frontend**

**spec:**

***# comment or delete the following line if you want to use a LoadBalancer***

**type: NodePort**

***# if your cluster supports it, uncomment the following to automatically create***

***# an external load-balanced IP for the frontend service.***

***# type: LoadBalancer***

**ports:**

**- port: 80**

**selector:**

**app: guestbook**

**tier: frontend**

**kubectl apply -f https://k8s.io/examples/application/guestbook/frontend-service.yaml**

Query the list of Services to verify that the frontend Service is running:

**kubectl get services**

Viewing the Frontend Service via **NodePort**

If you deployed this application to Minikube or a local cluster, you need to find the IP address to view your Guestbook.

1. Run the following command to get the IP address for the frontend Service.

**minikube service frontend --url**

Viewing the Frontend Service via **LoadBalancer**

If you deployed the **frontend-service.yaml** manifest with type: **LoadBalancer** you need to find the IP address to view your Guestbook.

1. Run the following command to get the IP address for the frontend Service.

**kubectl get service frontend**

Scale the Web Frontend

Scaling up or down is easy because your servers are defined as a Service that uses a Deployment controller.

1. Run the following command to scale up the number of frontend Pods:

**kubectl scale deployment frontend --replicas=5**

1. Query the list of Pods to verify the number of frontend Pods running:

**kubectl get pods**

1. Run the following command to scale down the number of frontend Pods:

**kubectl scale deployment frontend --replicas=2**

1. Query the list of Pods to verify the number of frontend Pods running:

**kubectl get pods**

## Cleaning up

Deleting the Deployments and Services also deletes any running Pods. Use labels to delete multiple resources with one command.

1. Run the following commands to delete all Pods, Deployments, and Services.
2. **kubectl delete deployment -l app=redis**
3. **kubectl delete service -l app=redis**
4. **kubectl delete deployment -l app=guestbook**

**kubectl delete service -l app=guestbook**

StatefulSet is the workload API object used to manage stateful applications.

Services

==========

* [NAT](https://en.wikipedia.org/wiki/Network_address_translation): network address translation
* [Source NAT](https://en.wikipedia.org/wiki/Network_address_translation#SNAT): replacing the source IP on a packet, usually with a node’s IP
* [Destination NAT](https://en.wikipedia.org/wiki/Network_address_translation#DNAT): replacing the destination IP on a packet, usually with a pod IP
* [VIP](https://kubernetes.io/docs/concepts/services-networking/service/#virtual-ips-and-service-proxies): a virtual IP, such as the one assigned to every Kubernetes Service
* [Kube-proxy](https://kubernetes.io/docs/concepts/services-networking/service/#virtual-ips-and-service-proxies): a network daemon that orchestrates Service VIP management on every node

## Source IP for Services with Type=ClusterIP

* Packets sent to ClusterIP from within the cluster are never source NAT’d if you’re running kube-proxy in [iptables mode](https://kubernetes.io/docs/concepts/services-networking/service/" \l "proxy-mode-iptables), which is the default since Kubernetes 1.2. Kube-proxy exposes its mode through a **proxyMode** endpoint:

**$ kubectl run source-ip-app --image=k8s.gcr.io/echoserver:1.4**

**deployment.apps/source-ip-app created**

**kubectl expose deployment source-ip-app --name=clusterip --port=80 --target-port=8080**

**service/clusterip exposed**

**kubectl get svc clusterip**

**kubectl run busybox -it --image=busybox --restart=Never --rm**

Source IP for Services with Type=NodePort

As of Kubernetes 1.5, packets sent to Services with [Type=NodePort](https://kubernetes.io/docs/concepts/services-networking/service/#nodeport) are source NAT’d by default. You can test this by creating a **NodePort** Service:

**kubectl expose deployment source-ip-app --name=nodeport --port=80 --target-port=8080 --type=NodePort**

**service/nodeport exposed**

**$ NODEPORT=$(kubectl get -o jsonpath="{.spec.ports[0].nodePort}" services nodeport)**

**$ NODES=$(kubectl get nodes -o jsonpath='{ $.items[\*].status.addresses[?(@.type=="ExternalIP")].address }')**

Source IP for Services with Type=LoadBalancer

As of Kubernetes 1.5, packets sent to Services with [Type=LoadBalancer](https://kubernetes.io/docs/concepts/services-networking/service/#loadbalancer) are source NAT’d by default, because all schedulable Kubernetes nodes in the **Ready** state are eligible for loadbalanced traffic. So if packets arrive at a node without an endpoint, the system proxies it to a node *with* an endpoint, replacing the source IP on the packet with the IP of the node (as described in the previous section).

You can test this by exposing the source-ip-app through a loadbalancer

**$ kubectl expose deployment source-ip-app --name=loadbalancer --port=80 --target-port=8080 --type=LoadBalancer**

**service/loadbalancer exposed**

**$ kubectl get svc loadbalancer**

**$ kubectl patch svc loadbalancer -p '{"spec":{"externalTrafficPolicy":"Local"}}'**

**$ kubectl get svc loadbalancer -o yaml | grep -i healthCheckNodePort**

**healthCheckNodePort: 32122**

**$ kubectl get pod -o wide -l run=source-ip-app**

Delete the Services:

**$ kubectl delete svc -l run=source-ip-app**

Delete the Deployment, ReplicaSet and Pod:

**$ kubectl delete deployment source-ip-app**

AppArmor is a Linux kernel security module that supplements the standard Linux user and group based permissions to confine programs to a limited set of resources.

To ensure that your Pods are receiving the expected protections, it is important to verify the Kubelet version of your nodes:

**kubectl get nodes -o=jsonpath=$'{range .items[\*]}{@.metadata.name}: {@.status.nodeInfo.kubeletVersion}\n{end}'**

**[Tutorials](https://kubernetes.io/docs/tutorials/)**

[Hello Minikube](https://kubernetes.io/docs/tutorials/hello-minikube/)

Learn Kubernetes Basics

[Learn Kubernetes Basics](https://kubernetes.io/docs/tutorials/kubernetes-basics/)

Create a Cluster

[Using Minikube to Create a Cluster](https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/cluster-intro/)

[Interactive Tutorial - Creating a Cluster](https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/cluster-interactive/)

Deploy an App

[Using kubectl to Create a Deployment](https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-app/deploy-intro/)

[Interactive Tutorial - Deploying an App](https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-app/deploy-interactive/)

Explore Your App

[Viewing Pods and Nodes](https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/)

[Interactive Tutorial - Exploring Your App](https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-interactive/)

Expose Your App Publicly

[Using a Service to Expose Your App](https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/expose-intro/)

[Interactive Tutorial - Exposing Your App](https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/expose-interactive/)

Scale Your App

[Running Multiple Instances of Your App](https://kubernetes.io/docs/tutorials/kubernetes-basics/scale/scale-intro/)

[Interactive Tutorial - Scaling Your App](https://kubernetes.io/docs/tutorials/kubernetes-basics/scale/scale-interactive/)

Update Your App

[Performing a Rolling Update](https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/)

[Interactive Tutorial - Updating Your App](https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-interactive/)

Online Training Courses

[Overview of Kubernetes Online Training](https://kubernetes.io/docs/tutorials/online-training/overview/)

Configuration

[Configuring Redis using a ConfigMap](https://kubernetes.io/docs/tutorials/configuration/configure-redis-using-configmap/)

Stateless Applications

[Exposing an External IP Address to Access an Application in a Cluster](https://kubernetes.io/docs/tutorials/stateless-application/expose-external-ip-address/)

[Example: Deploying PHP Guestbook application with Redis](https://kubernetes.io/docs/tutorials/stateless-application/guestbook/)

Stateful Applications

[StatefulSet Basics](https://kubernetes.io/docs/tutorials/stateful-application/basic-stateful-set/)

[Example: Deploying WordPress and MySQL with Persistent Volumes](https://kubernetes.io/docs/tutorials/stateful-application/mysql-wordpress-persistent-volume/)

[Example: Deploying Cassandra with Stateful Sets](https://kubernetes.io/docs/tutorials/stateful-application/cassandra/)

[Running ZooKeeper, A Distributed System Coordinator](https://kubernetes.io/docs/tutorials/stateful-application/zookeeper/)

Clusters

[AppArmor](https://kubernetes.io/docs/tutorials/clusters/apparmor/)

Services

[Using Source IP](https://kubernetes.io/docs/tutorials/services/source-ip/)

[Edit This Page](https://github.com/kubernetes/website/edit/master/content/en/docs/tutorials/clusters/apparmor.md)

AppArmor

**FEATURE STATE:** **Kubernetes v1.4** [beta](https://kubernetes.io/docs/tutorials/clusters/apparmor/)

AppArmor is a Linux kernel security module that supplements the standard Linux user and group based permissions to confine programs to a limited set of resources. AppArmor can be configured for any application to reduce its potential attack surface and provide greater in-depth defense. It is configured through profiles tuned to whitelist the access needed by a specific program or container, such as Linux capabilities, network access, file permissions, etc. Each profile can be run in either *enforcing* mode, which blocks access to disallowed resources, or *complain* mode, which only reports violations.

AppArmor can help you to run a more secure deployment by restricting what containers are allowed to do, and/or provide better auditing through system logs. However, it is important to keep in mind that AppArmor is not a silver bullet and can only do so much to protect against exploits in your application code. It is important to provide good, restrictive profiles, and harden your applications and cluster from other angles as well.

* [**Objectives**](https://kubernetes.io/docs/tutorials/clusters/apparmor/#objectives)
* [**Before you begin**](https://kubernetes.io/docs/tutorials/clusters/apparmor/#before-you-begin)
* [**Securing a Pod**](https://kubernetes.io/docs/tutorials/clusters/apparmor/#securing-a-pod)
* [**Example**](https://kubernetes.io/docs/tutorials/clusters/apparmor/#example)
* [**Administration**](https://kubernetes.io/docs/tutorials/clusters/apparmor/#administration)
* [**Authoring Profiles**](https://kubernetes.io/docs/tutorials/clusters/apparmor/#authoring-profiles)
* [**API Reference**](https://kubernetes.io/docs/tutorials/clusters/apparmor/#api-reference)
* [**What's next**](https://kubernetes.io/docs/tutorials/clusters/apparmor/#what-s-next)

Objectives

* See an example of how to load a profile on a node
* Learn how to enforce the profile on a Pod
* Learn how to check that the profile is loaded
* See what happens when a profile is violated
* See what happens when a profile cannot be loaded

Before you begin

Make sure:

1. Kubernetes version is at least v1.4 – Kubernetes support for AppArmor was added in v1.4. Kubernetes components older than v1.4 are not aware of the new AppArmor annotations, and will **silently ignore**any AppArmor settings that are provided. To ensure that your Pods are receiving the expected protections, it is important to verify the Kubelet version of your nodes:

**$ kubectl get nodes -o=jsonpath=$'{range .items[\*]}{@.metadata.name}: {@.status.nodeInfo.kubeletVersion}\n{end}'**

**gke-test-default-pool-239f5d02-gyn2: v1.4.0**

**gke-test-default-pool-239f5d02-x1kf: v1.4.0**

**gke-test-default-pool-239f5d02-xwux: v1.4.0**

1. AppArmor kernel module is enabled – For the Linux kernel to enforce an AppArmor profile, the AppArmor kernel module must be installed and enabled. Several distributions enable the module by default, such as Ubuntu and SUSE, and many others provide optional support. To check whether the module is enabled, check the **/sys/module/apparmor/parameters/enabled** file:

**$ cat /sys/module/apparmor/parameters/enabled**

**Y**

If the Kubelet contains AppArmor support (>= v1.4), it will refuse to run a Pod with AppArmor options if the kernel module is not enabled.

**Note:** Ubuntu carries many AppArmor patches that have not been merged into the upstream Linux kernel, including patches that add additional hooks and features. Kubernetes has only been tested with the upstream version, and does not promise support for other features.

1. Container runtime is Docker – Currently the only Kubernetes-supported container runtime that also supports AppArmor is Docker. As more runtimes add AppArmor support, the options will be expanded. You can verify that your nodes are running docker with:

**$ kubectl get nodes -o=jsonpath=$'{range .items[\*]}{@.metadata.name}: {@.status.nodeInfo.containerRuntimeVersion}\n{end}'**

**gke-test-default-pool-239f5d02-gyn2: docker://1.11.2**

**gke-test-default-pool-239f5d02-x1kf: docker://1.11.2**

**gke-test-default-pool-239f5d02-xwux: docker://1.11.2**

If the Kubelet contains AppArmor support (>= v1.4), it will refuse to run a Pod with AppArmor options if the runtime is not Docker.

1. Profile is loaded – AppArmor is applied to a Pod by specifying an AppArmor profile that each container should be run with. If any of the specified profiles is not already loaded in the kernel, the Kubelet (>= v1.4) will reject the Pod. You can view which profiles are loaded on a node by checking the **/sys/kernel/security/apparmor/profiles** file. For example:

**$ ssh gke-test-default-pool-239f5d02-gyn2 "sudo cat /sys/kernel/security/apparmor/profiles | sort"**

**apparmor-test-deny-write (enforce)**

**apparmor-test-audit-write (enforce)**

**docker-default (enforce)**

**k8s-nginx (enforce)**

For more details on loading profiles on nodes, see [Setting up nodes with profiles](https://kubernetes.io/docs/tutorials/clusters/apparmor/#setting-up-nodes-with-profiles).

As long as the Kubelet version includes AppArmor support (>= v1.4), the Kubelet will reject a Pod with AppArmor options if any of the prerequisites are not met. You can also verify AppArmor support on nodes by checking the node ready condition message (though this is likely to be removed in a later release):

**$ kubectl get nodes -o=jsonpath=$'{range .items[\*]}{@.metadata.name}: {.status.conditions[?(@.reason=="KubeletReady")].message}\n{end}'**

**gke-test-default-pool-239f5d02-gyn2: kubelet is posting ready status. AppArmor enabled**

**gke-test-default-pool-239f5d02-x1kf: kubelet is posting ready status. AppArmor enabled**

**gke-test-default-pool-239f5d02-xwux: kubelet is posting ready status. AppArmor enabled**

Securing a Pod

 AppArmor is currently in beta, so options are specified as annotations. Once support graduates to general availability, the annotations will be replaced with first-class fields

AppArmor profiles are specified per-container. To specify the AppArmor profile to run a Pod container with, add an annotation to the Pod’s metadata:

**kubectl get events | grep Created**