RX-M Cloud Native Consulting

Kubernetes

Lab 8 – Metrics

Starting from Kubernetes 1.8, resource usage metrics, such as container CPU and memory usage, are available in Kubernetes through the Metrics API. These metrics can be either accessed directly by a user with `kubectl top`, or used by a controller in the cluster, such as the Horizontal Pod Autoscaler.

The Metrics API reports the current resource utilization of a node or pod or set thereof. The metrics API is designed to expose the current resource utilization only. Those wishing to view a history of resource utilization should use a monitoring tool and database such as Prometheus or InfluxDB/Grafana.

Kubernetes 1.7 introduced the aggregator API feature. This allows add on services to extend the base K8s API. The Metrics API uses the aggregator feature and exposes an API available at /apis/metrics.k8s.io/, offering the same security, scalability and reliability guarantees as the rest of the api-server hosted end points.

The Metrics server was first released in September 2017 and is not installed automatically by all K8s installers. The Metrics Server does not run as a module within the Controller Manager, rather it runs as a stand alone deployment, typically in the kube-system namespace. The Metrics Server requires an SA with appropriate RBAC roles so that it can communicate with the api-server and extend the K8s API.

The Metrics Server must also be equipped with certificates/keys that will allow it to connect to each of the Kubelets in the system. Kubelets expose a `/metrics` end point, the Metrics Server scrapes this endpoint for metrics data regularly, storing the results in memory. When in bound requests for metrics arrive at the api-server the metrics server is called to answer them.

In this lab we'll install the Metrics Server and explore the metrics enabled features of Kubernetes. Delete all user defined deployments, services, and other resources before starting.

1. Explore the Metrics Server repository

Basic Metrics Server deployment manifests are available in the /deploy directory in the Metrics Server source repo:

https://github.com/kubernetes-incubator/metrics-server

The Metrics Server repository includes a set of manifests that create all of the cluster resources needed to run the Metrics Server. Clone the Metrics Server repository from GitHub:

```
user@ubuntu:~$ git clone --depth 1 https://github.com/kubernetes-incubator/metrics-server

Cloning into 'metrics-server'...
remote: Enumerating objects: 88, done.
remote: Counting objects: 100% (88/88), done.
remote: Compressing objects: 100% (74/74), done.
remote: Total 88 (delta 11), reused 39 (delta 5), pack-reused 0
Unpacking objects: 100% (88/88), done.
Checking connectivity... done.

user@ubuntu:~$
```

Now change into the repository directory and display the manifests in the deploy/1.8+ directory:

```
  user@ubuntu:~$ cd metrics-server/

user@ubuntu:~/metrics-server$ ls -l

total 152
drwxrwxr-x  3 user user  4096 Jan 24 15:44 cmd
-rw-rw-r--  1 user user   148 Jan 24 15:44 code-of-conduct.md
-rw-rw-r--  1 user user   418 Jan 24 15:44 CONTRIBUTING.md
drwxrwxr-x  6 user user  4096 Jan 24 15:44 deploy
```

```
-rw-rw-r--  1 user user  3364 Jan 24 15:44 go.mod
-rw-rw-r--  1 user user 84975 Jan 24 15:44 go.sum
drwxrwxr-x  2 user user  4096 Jan 24 15:44 hack
-rw-rw-r--  1 user user 11357 Jan 24 15:44 LICENSE
-rw-rw-r--  1 user user  4020 Jan 24 15:44 Makefile
-rw-rw-r--  1 user user   201 Jan 24 15:44 OWNERS
-rw-rw-r--  1 user user   177 Jan 24 15:44 OWNERS_ALIASES
drwxrwxr-x 11 user user  4096 Jan 24 15:44 pkg
-rw-rw-r--  1 user user  5305 Jan 24 15:44 README.md
-rw-rw-r--  1 user user   541 Jan 24 15:44 SECURITY_CONTACTS
drwxrwxr-x  2 user user  4096 Jan 24 15:44 test

user@ubuntu:~/metrics-server$ ls -l deploy/

total 16
drwxrwxr-x 2 user user 4096 Jan 24 15:44 1.7
drwxrwxr-x 2 user user 4096 Jan 24 15:44 1.8+
drwxrwxr-x 2 user user 4096 Jan 24 15:44 docker
drwxrwxr-x 2 user user 4096 Jan 24 15:44 minikube

user@ubuntu:~/metrics-server$ ls -l deploy/1.8+/

total 28
-rw-rw-r-- 1 user user  397 Jan 24 15:44 aggregated-metrics-reader.yaml
-rw-rw-r-- 1 user user  303 Jan 24 15:44 auth-delegator.yaml
-rw-rw-r-- 1 user user  324 Jan 24 15:44 auth-reader.yaml
-rw-rw-r-- 1 user user  298 Jan 24 15:44 metrics-apiservice.yaml
-rw-rw-r-- 1 user user 1183 Jan 24 15:44 metrics-server-deployment.yaml
-rw-rw-r-- 1 user user  297 Jan 24 15:44 metrics-server-service.yaml
-rw-rw-r-- 1 user user  532 Jan 24 15:44 resource-reader.yaml

user@ubuntu:~/metrics-server$
```

The Metrics Server deployment uses sever manifests. Generate a listing of the types of resources that will be created:

```
user@ubuntu:~/metrics-server$ kubectl apply -f deploy/1.8+/ --dry-run

clusterrole.rbac.authorization.k8s.io/system:aggregated-metrics-reader created (dry run)
clusterrolebinding.rbac.authorization.k8s.io/metrics-server:system:auth-delegator created (dry
run)
rolebinding.rbac.authorization.k8s.io/metrics-server-auth-reader created (dry run)
apiservice.apiregistration.k8s.io/v1beta1.metrics.k8s.io created (dry run)
serviceaccount/metrics-server created (dry run)
deployment.apps/metrics-server created (dry run)
service/metrics-server created (dry run)
clusterrole.rbac.authorization.k8s.io/system:metrics-server created (dry run)
clusterrolebinding.rbac.authorization.k8s.io/system:metrics-server created (dry run)

user@ubuntu:~/metrics-server$
```

Though a range of resources are defined, you can see that most are RBAC elements:

- deploy/1.8+/aggregated-metrics-reader.yaml:kind: ClusterRole
- deploy/1.8+/auth-delegator.yaml:kind: ClusterRoleBinding
- deploy/1.8+/auth-reader.yaml:kind: RoleBinding
- deploy/1.8+/metrics-server-deployment.yaml:kind: ServiceAccount
- deploy/1.8+/resource-reader.yaml:kind: ClusterRole
- deploy/1.8+/resource-reader.yaml:kind: ClusterRoleBinding

The manifests create two ClusterRoles. Display the rules defined:

```
user@ubuntu:~/metrics-server$ grep rules -A21 deploy/1.8+/*

deploy/1.8+/aggregated-metrics-reader.yaml:rules:
deploy/1.8+/aggregated-metrics-reader.yaml-- apiGroups: ["metrics.k8s.io"]
deploy/1.8+/aggregated-metrics-reader.yaml-  resources: ["pods", "nodes"]
```

```
deploy/1.8+/aggregated-metrics-reader.yaml-  verbs: ["get", "list", "watch"]
--
deploy/1.8+/resource-reader.yaml:rules:
deploy/1.8+/resource-reader.yaml-- apiGroups:
deploy/1.8+/resource-reader.yaml-  - ""
deploy/1.8+/resource-reader.yaml-  resources:
deploy/1.8+/resource-reader.yaml-  - pods
deploy/1.8+/resource-reader.yaml-  - nodes
deploy/1.8+/resource-reader.yaml-  - nodes/stats
deploy/1.8+/resource-reader.yaml-  - namespaces
deploy/1.8+/resource-reader.yaml-  - configmaps
deploy/1.8+/resource-reader.yaml-  verbs:
deploy/1.8+/resource-reader.yaml-  - get
deploy/1.8+/resource-reader.yaml-  - list
deploy/1.8+/resource-reader.yaml-  - watch
deploy/1.8+/resource-reader.yaml----
deploy/1.8+/resource-reader.yaml-apiVersion: rbac.authorization.k8s.io/v1
deploy/1.8+/resource-reader.yaml-kind: ClusterRoleBinding
deploy/1.8+/resource-reader.yaml-metadata:
deploy/1.8+/resource-reader.yaml-  name: system:metrics-server
deploy/1.8+/resource-reader.yaml-roleRef:
deploy/1.8+/resource-reader.yaml-  apiGroup: rbac.authorization.k8s.io
deploy/1.8+/resource-reader.yaml-  kind: ClusterRole
deploy/1.8+/resource-reader.yaml-  name: system:metrics-server

user@ubuntu:~/metrics-server$
```

The aggregated-metrics-reader provides access to the pod information within the metrics.k8s.io API group. The resource-reader provides access to information methods for pods, nodes, nodes/stats and namespaces across all API groups and information access to deployments within the extensions API group.

The ClusterRoleBinding objects bind the ClusterRoles to the metrics-server ServiceAccount created in the kube-system namespace. The one RoleBinding binds the preexisting extension-apiserver-authentication-reader Role to the Metric Server's service account:

```
user@ubuntu:~/metrics-server$ cat deploy/1.8+/auth-reader.yaml
```

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: metrics-server-auth-reader
  namespace: kube-system
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: extension-apiserver-authentication-reader
subjects:
- kind: ServiceAccount
  name: metrics-server
  namespace: kube-system
```

```
user@ubuntu:~/metrics-server$ kubectl get role -n kube-system extension-apiserver-
authentication-reader

NAME                                        AGE
extension-apiserver-authentication-reader   4h25m

user@ubuntu:~/metrics-server$
```

The extension-apiserver-authentication-reader allows the addon API server to read the client CA file from the extension-apiserver-authentication ConfigMap in the kube-system namespace. This is required in order for the Metrics addon API to delegate authentication and authorization requests to the main Kubernetes API server.

The remaining three manifests support the Metrics Server operation:

- metrics-apiservice.yaml:kind: APIService

- metrics-server-deployment.yaml:kind: Deployment
- metrics-server-service.yaml:kind: Service

The APIService manifest defines the metric-server service API extension to the core Kubernetes API:

```
user@ubuntu:~/metrics-server$ cat deploy/1.8+/metrics-apiservice.yaml
```

```
---
apiVersion: apiregistration.k8s.io/v1beta1
kind: APIService
metadata:
  name: v1beta1.metrics.k8s.io
spec:
  service:
    name: metrics-server
    namespace: kube-system
  group: metrics.k8s.io
  version: v1beta1
  insecureSkipTLSVerify: true
  groupPriorityMinimum: 100
  versionPriority: 100
```

```
user@ubuntu:~/metrics-server$
```

Ad hoc API extensions make use of the Kubernetes API aggregation function. Kubernetes API aggregation enables the dynamic installation of additional Kubernetes-style APIs in a cluster. These can either be pre-built, existing 3rd party solutions, such as service-catalog, or user-created APIs like apiserver-builder.

The aggregation layer runs in-process with the kube-apiserver. Until an extension resource is registered, the aggregation layer will do nothing. To register an API an APIService object is used to "claim" a URL path in the Kubernetes API. At this point, the aggregation layer will proxy anything sent to that API path to the registered APIService.

In this case the metrics-server claims the `/apis/metrics.k8s.io/` path.

The metrics server deployment is fairly simple and just runs the k8s.gcr.io/metrics-server-amd64:v0.3.6 image. The Metrics Server service is also simple, creating a "metrics-server" service that accepts traffic on port 443.

Examine any of the yaml manifests you find interesting.

2. Deploy the Metrics Server

> N.B. If you have a multi-node cluster, make sure you deploy the metrics server to your master node. Use `kubectl cordon <worker node>` and remove the master taint if you have it with `kubectl taint nodes $(hostname) node-role.kubernetes.io/master-` to ensure this lab's pod run on your master node.

We can launch the Metrics Server by creating all of the resources in the deploy/1.8+ directory by providing `kubectl apply -f` with the directory path. Try it:

```
user@ubuntu:~/metrics-server$ kubectl apply -f deploy/1.8+/

clusterrole.rbac.authorization.k8s.io/system:aggregated-metrics-reader created
clusterrolebinding.rbac.authorization.k8s.io/metrics-server:system:auth-delegator created
rolebinding.rbac.authorization.k8s.io/metrics-server-auth-reader created
apiservice.apiregistration.k8s.io/v1beta1.metrics.k8s.io created
serviceaccount/metrics-server created
deployment.apps/metrics-server created
service/metrics-server created
clusterrole.rbac.authorization.k8s.io/system:metrics-server created
clusterrolebinding.rbac.authorization.k8s.io/system:metrics-server created

user@ubuntu:~/metrics-server$
```

```
user@ubuntu:~/metrics-server$ kubectl get all -n kube-system -l k8s-app=metrics-server

NAME                                  READY   STATUS    RESTARTS   AGE
pod/metrics-server-694db48df9-lzm99   1/1     Running   0          43s

NAME                             READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/metrics-server   1/1     1            1           43s

NAME                                        DESIRED   CURRENT   READY   AGE
replicaset.apps/metrics-server-694db48df9   1         1         1       43s

user@ubuntu:~/metrics-server$
```

Great, the Metrics Server is deployed and running! Now try a metrics api request:

```
user@ubuntu:~/metrics-server$ curl -k https://localhost:6443/apis/metrics.k8s.io/
```

```
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {

  },
  "status": "Failure",
  "message": "forbidden: User \"system:anonymous\" cannot get path \"/apis/metrics.k8s.io/\"",
  "reason": "Forbidden",
  "details": {

  },
  "code": 403
}
```

```
user@ubuntu:~/metrics-server$
```

We cannot connect to the apiserver without a user identity, let's use an existing one:

```
user@ubuntu:~/metrics-server$ kubectl get sa

NAME      SECRETS   AGE
default   1         24h

user@ubuntu:~/metrics-server$
```

Get the token and certificate from the ServiceAccount's token secret for use in your API requests.

This will require `jq` for JSON parsing on the command line, install it:

```
user@ubuntu:~/metrics-server$ sudo apt-get install jq -y

...

user@ubuntu:~/metrics-server$
```

Start by setting the `SERVICE_ACCOUNT` variable:

```
user@ubuntu:~/metrics-server$ SERVICE_ACCOUNT=default

user@ubuntu:~/metrics-server$
```

Get the ServiceAccount's token Secret's name:

```
user@ubuntu:~/metrics-server$ SECRET=$(kubectl get serviceaccount ${SERVICE_ACCOUNT} -o json \
| jq -Mr '.secrets[].name | select(contains("token"))') && echo $SECRET

default-token-ksft2

user@ubuntu:~/metrics-server$
```

Extract the Bearer token from the Secret and decode:

```
user@ubuntu:~/metrics-server$ TOKEN=$(kubectl get secret ${SECRET} -o json \
| jq -Mr '.data.token' | base64 -d) && echo $TOKEN

eyJhbGciOiJSUzI1NiIsImtpZCI6IjRxQmtKN1g3ZlhWT1hFb0ItTVRYRVo5TGRZVzU0TGlwLXBtRWRVdWdhU0kifQ.eyJpc
3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiO
iJkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZWNyZXQubmFtZSI6ImRlZmF1bHQtdG9rZW4ta3Nmd
DIiLCJrdWJlcm5ldGVzLmlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC5uYW1lIjoiZGVmYXVsdCIsImt1YmVyb
mV0ZXMuaW8vc2VydmljZWFjY291bnQvc2VydmljZS1hY2NvdW50LnVpZCI6ImMwOTk2Y2ZmLTcwZDAtNGNlMy1iMTlmLWQ1Z
TliMmQ2YWRmMCIsInN1YiI6InN5c3RlbTpzZXJ2aWNlYWNjb3VudDpkZWZhdWx0OmRlZmF1bHQifQ.Fpnx58zm4By_IAjrU4
7um1AgxxjbK4cnbjZTgXlDc1Pbv-
n9SUZs_lj0hZTLwXNyTx1IVfntinfSBY6g7i3MEsBcUs6Q6zCYAC_vRZKi6mkamqT_93cM8vSjrPqgZ9k8X9UOkYxD_kUbML
EWpzXjJiAenGuBMBnDplAK70qQUV2SO2JAf7GyOMF79KOqkiTaiNLNQmRrZoyOwmYCFWUTwgMrUkruT9gow2eyk7xhIYiUzf
iG-C7v0QLXyoI_2pyNbYbULibhExbyg2S-dXfCfJmOuHNPt0G8hyXosHRQmVDI875rX-0SQJVQ01AOmPUxEaerc9S-
ayL9ROPGlfA38g

user@ubuntu:~/metrics-server$
```

Extract, decode and write the ca.crt to a temporary location:

```
user@ubuntu:~/metrics-server$ kubectl get secret ${SECRET} -o json \
| jq -Mr '.data["ca.crt"]' | base64 -d > /tmp/ca.crt

user@ubuntu:~/metrics-server$
```

Now retry the metrics api request once more, using your new identity:

```
user@ubuntu:~/metrics-server$ curl -k https://localhost:6443/apis/metrics.k8s.io/ \
--header "Authorization: Bearer $TOKEN" --cacert /tmp/ca.crt
```


```
{
  "kind": "APIGroup",
  "apiVersion": "v1",
  "name": "metrics.k8s.io",
  "versions": [
    {
      "groupVersion": "metrics.k8s.io/v1beta1",
      "version": "v1beta1"
    }
  ],
  "preferredVersion": {
    "groupVersion": "metrics.k8s.io/v1beta1",
    "version": "v1beta1"
  }
}
```


```
user@ubuntu:~/metrics-server$
```

Great, the Metrics Server is running *and* we can query it!

3. Test the Metrics Server

The `kubectl top` command can be used to display metrics for pods or nodes. Try it:

```
user@ubuntu:~/metrics-server$ kubectl top node

error: metrics not available yet

user@ubuntu:~/metrics-server$
```

Hmm, no metrics. Try checking the metrics server log to see if there are any issues indicated by the metrics server pod:

```
user@ubuntu:~/metrics-server$ kubectl get pods -n kube-system -l k8s-app=metrics-server

NAME                              READY   STATUS    RESTARTS   AGE
metrics-server-694db48df9-lzm99   1/1     Running   0          78s

user@ubuntu:~/metrics-server$ kubectl -n kube-system logs metrics-server-694db48df9-lzm99

I0124 23:49:33.028127       1 serving.go:312] Generated self-signed cert (/tmp/apiserver.crt,
/tmp/apiserver.key)
I0124 23:49:33.303878       1 secure_serving.go:116] Serving securely on [::]:4443
E0124 23:50:45.557373       1 manager.go:111] unable to fully collect metrics: unable to fully
scrape metrics from source kubelet_summary:ubuntu: unable to fetch metrics from Kubelet ubuntu
(ubuntu): Get https://ubuntu:10250/stats/summary?only_cpu_and_memory=true: dial tcp
66.96.162.149:10250: connect: connection refused
E0124 23:51:19.105292       1 reststorage.go:135] unable to fetch node metrics for node
"ubuntu": no metrics known for node
E0124 23:51:45.520002       1 manager.go:111] unable to fully collect metrics: unable to fully
scrape metrics from source kubelet_summary:ubuntu: unable to fetch metrics from Kubelet ubuntu
(ubuntu): Get https://ubuntu:10250/stats/summary?only_cpu_and_memory=true: dial tcp
66.96.162.149:10250: connect: connection refused

user@ubuntu:~/metrics-server$
```

One issue here is that the metrics server cannot communicate with any of the kubelets for this host. In the example above, the hostname "ubuntu" is routing to an IP address that is rejecting the connection, meaning the DNS configuration in this environment is not configurd correctly.

Even if it could reach the host, another problem is that the Metrics server requests its metrics from the Kubelets but rejects the kubelet responses. This is because the Metrics Server attempts to use the api-server CA cert to verify the Kubelet's response. In the Kubeadm case today the Kubelet uses an api-server CA signed cert to reach out to the api-server but when listening on its own API port (where the /metrics endpoint we want is located) is uses a self signed certificate. This causes the Metrics Server to reject the TLS session upon connection. Existing workarounds will suggest using `--kubelet-insecure-tls` to tell the Metrics Server to not check the Kubelet's TLS cert. This approach is not ideal for production deployments.

3a. Force the kubelet to request a new certificate from the cluster

To fix this you will need to regenerate the kubelet's certificate using the cluster CA certificate. This is possible by forcing the node kubelet to go through the TLS bootstrap process again to request a new serving certificate. The serving certificate bears the cluster CA's signature and also includes the hostname and IP addresses of the kubelet as subject alternative names. Any services that need to verify the the kubelet's identities will be able to do so using a new serving certificate.

To begin, retrieve the bootstrap token from the cluster. This token is what your node's kubelet used initially to communicate with the API server. This bootstrap token is assigned to the system:bootstrappers group, which gives it all the permissions it needs to connect to the API server and request a certificate. These tokens are backed by secrets in the cluster.

Retrieve the bootstrap token from the cluster's secrets:

```
user@ubuntu:~/metrics-server$ kubectl get secrets -n kube-system | grep bootstrap

bootstrap-signer-token-mnmtt                     kubernetes.io/service-account-token   3
24h
bootstrap-token-pllvtu                           bootstrap.kubernetes.io/token         7
```

```
24h

user@ubuntu:~/metrics-server$ export BOOTSTRAPSECRET=bootstrap-token-pllvtu

user@ubuntu:~/metrics-server$ kubectl get secret -n kube-system $BOOTSTRAPSECRET -o yaml

apiVersion: v1
data:
  auth-extra-groups: c3lzdGVtOmJvb3RzdHJhcHBlcnM6a3ViZWFkbTpkZWZhdWx0LW5vZGUtdG9rZW4=
  expiration: MjAyMC0wMS0yNVQxNjoxMDoyMi0wODowMA==
  token-id: YzJpaGo1
  token-secret: bDZhOG5qOHJkeHBsZXpzOQ==
  usage-bootstrap-authentication: dHJ1ZQ==
  usage-bootstrap-signing: dHJ1ZQ==
kind: Secret
metadata:
  creationTimestamp: "2020-01-25T00:10:22Z"
  name: bootstrap-token-c2ihj5
  namespace: kube-system
  resourceVersion: "26242"
  selfLink: /api/v1/namespaces/kube-system/secrets/bootstrap-token-c2ihj5
  uid: 082db190-155c-4a3f-8d9d-e62a2080427d
type: bootstrap.kubernetes.io/token

user@ubuntu:~/metrics-server$
```

The token-id and token-secret are needed to present a complete bookstrap token.

Using `-o jsonpath` to extract specific values from the secret, retrieve the `token-id` and the `token-secret`. To output the values in plain text, pipe the output into `base64 --decode`:

```
user@ubuntu:~/metrics-server$ kubectl get secret -n kube-system $BOOTSTRAPSECRET -o
jsonpath='{.data.token-id}' \
| base64 --decode && echo

c2ihj5

user@ubuntu:~/metrics-server$ kubectl get secret -n kube-system $BOOTSTRAPSECRET -o
jsonpath='{.data.token-secret}' \
| base64 --decode && echo

l6a8nj8rdxplezs9

user@ubuntu:~/metrics-server$
```

The proper format for a token is `token-id.token-secret`. When properly formatted, the complete bootstrap token for this cluster is: `c2ihj5.l6a8nj8rdxplezs9`

> N.B. In Kubeadm clusters like ours, bootstrap tokens are only valid for 24 hours. If your bootstrap token is over 24 hours old, you can use kubeadm to create a new token with `kubeadm token create`. This command outputs a complete bootstrap token to the terminal and creates a corresponding secret in the cluster.
>
> ```
> user@ubuntu:~$ kubeadm token create
>
> ezgumy.l3p5h3hn4xsd9u2k
>
> user@ubuntu:~$
> ```
>
> If this is how you retrieved the bootstrap token, make sure to use whatever output is put in the terminal for the following steps.

Now you need to create a bootstrap kubelet configuration. This configuration tells the Kubelet to re-register itself with the cluster. First you need to see the cluster's connection information by reading the current kubelet configurtion at `/etc/kubernetes/kubelet.conf`:

```
user@ubuntu:~/metrics-server$ sudo head /etc/kubernetes/kubelet.conf

apiVersion: v1
clusters:
- cluster:
    certificate-authority-data:
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUN5RENDQWJDZ0F3SUJBZ0lCQURBTkJna3Foa2lHOXcwQkFRc0ZBREFW
TVJNd0VRWURWUVFERXdwcmRXSmwKY201bGRHVnpNQjRYRFRJd01ERXlNekl6TkRVeU5Wb1hEVE13TURFeU1ESXpORFV5TlZv
d0ZURVRNQkVHQTFVRQpBeE1LYTNWaVpYSnVaWFJsY3pDQ0FTSXdEUVlKS29aSWh2Y05BUUVCQlFBRGdnRVBBRENDQVFvQ2dn
RUJBSjdJCmZNUGpQeUxTZWZldFQvZE9KYllaV1RKMWE0eEovSngzbFByV3ovcDFUcEVsUmZVemFwVFR5bUw2Tkk1MFd2VFAK
M1I3ZXdsakc0WWxZWXRKN0tKRkJTR2N4UFBrdzdHV3AxSUR1YnRrRjhtYTVXU2NNNHplelI0K3FEbmYySzZUNwpVNldscWt0
WEUzcDR2YXhBTERXNHNua0NPSlpZQXE1RlFWb1owQTR1VzU5bkY1ZFJ5RGNXMkw3dWRQYVJuQ3A2CmZIbmtPSUo4TUtsWjQ3
UHJkWFhCWGU1U0lYZURVZGtJUkwrRXdRbmpaWjhUL05MSkYwbXpEeTVrYzd3SHIrMTEKaStCWDdPaDNUYzFtUDlVaVRha2V1
cTA3M2JIN2c0TGFuR0ExRXkvK1VWZVIyTGg5V3grOUJJQjBSaFJQZ2FNdAozSldoL0I1ME9DU3FFVVMzZ204Q0F3RUFBYU1q
TUNFd0RnWURWUjBQQVFIL0JBUURBZ0trTUE4R0ExVWRFd0VCCi93UUZNQU1CQWY4d0RRWUpLb1pJaHZjTkFRUxCUUFEZ2dF
QkFIQVJPNkxwMEJ5NmpzTS9MWDRzaThITnJKTU8KdVlOejk5aTc0OXlHdEJOQjJHd1FJMHd5NzJLNGFvMWU2MG5XelhlYXcw
QTJBWldNbTIwVzZjM1ltc1JpV1dCNAp2WDB2UTU0cGE5YzE5dmliUTk4NXYwNmNmNnFnYVRZQzNjWWtuTDVxSnZsN1RwdUpV
QkQyUkhIVDFsS0lxSG5XCnpDVXhXZjdpVk55dmwxMFZmT1hFbUhYK0phN05jSTAraGR0ZHFqVFQxZ21MdmM4b0dUblFtN0tu
M0d2My9aRlMKem1yK0JSQjNIYVg1ZUJoVHVkRFp6WDNQYTZtc28yYmtsVUR4Y2FWcHB1bmVmdWZXNllTWFVuNHc0aE5yMFhW
Ywo3Qll0bGVVMXVFeTF0UlBrdjFaNHpYclpXTDByOXpVdGJ4b0JONUZ1V1B1K1ZoRlNPd0V3Q25BVDVSYz0KLS0tLS1FTkQg
Q0VSVElGSUNBVEUtLS0tLQo=
    server: https://192.168.229.132:6443
  name: default-cluster
contexts:
- context:
    cluster: default-cluster
    namespace: default

user@ubuntu:~/metrics-server$
```

We will use the value of the `server` key under `-cluster:` to inform the new bootstrap config how to connect to the cluster.

Now we can construct the bootstrap configuration using kubectl. First, add the cluster connection and certificate authority information to the file using `kubectl config set-cluster`. Be sure to substitute the `--server` flag with the value of `server:` from the kubelet.conf file:

```
user@ubuntu:~/metrics-server$ kubectl config set-cluster bootstrap \
--kubeconfig=bootstrap-kubelet.conf \
--certificate-authority='/etc/kubernetes/pki/ca.crt' \
--server='https://192.168.229.132:6443'


Cluster "bootstrap" set.

user@ubuntu:~/metrics-server$
```

Next, associate the bootstrap token you retrieved earlier to a user called `kubelet-bootstrap`. Be sure to substitute the value of `--token` with the token you retrieved or created earlier:

```
user@ubuntu:~/metrics-server$ kubectl config set-credentials kubelet-bootstrap \
--kubeconfig=bootstrap-kubelet.conf \
--token=c2ihj5.l6a8nj8rdxplezs9

User "kubelet-bootstrap" set.

user@ubuntu:~/metrics-server$
```

Now, set a context that associates the `kubelet-bootstrap` user to the cluster connection information:

```
ubuntu@ubuntu:~$ kubectl config set-context bootstrap \
--kubeconfig=bootstrap-kubelet.conf \
--user=kubelet-bootstrap --cluster=bootstrap

Context "bootstrap" created.
```

```
user@ubuntu:~/metrics-server$
```

Finally, create the file by switching to the bootstrap context with `kubectl config use-context`:

```
user@ubuntu:~/metrics-server$ kubectl config use-context bootstrap \
--kubeconfig=bootstrap-kubelet.conf

Switched to context "bootstrap".

user@ubuntu:~/metrics-server$
```

The file is now available. Use `cat` to view the contents.

```
user@ubuntu:~/metrics-server$ cat bootstrap-kubelet.conf

apiVersion: v1
clusters:
- cluster:
    certificate-authority: /etc/kubernetes/pki/ca.crt
    server: https://192.168.229.132:6443
  name: bootstrap
contexts:
- context:
    cluster: bootstrap
    user: kubelet-bootstrap
  name: bootstrap
current-context: bootstrap
kind: Config
preferences: {}
users:
- name: kubelet-bootstrap
  user:
    token: c2ihj5.l6a8nj8rdxplezs9

user@ubuntu:~/metrics-server$
```

Copy `bootstrap-kubelet.conf` to `/etc/kubernetes/`:

```
user@ubuntu:~/metrics-server$ sudo cp bootstrap-kubelet.conf /etc/kubernetes/bootstrap-
kubelet.conf

user@ubuntu:~/metrics-server$ sudo cat /etc/kubernetes/bootstrap-kubelet.conf

apiVersion: v1
clusters:
- cluster:
    certificate-authority: /etc/kubernetes/pki/ca.crt
    server: https://192.168.229.132:6443
  name: bootstrap
contexts:
- context:
    cluster: bootstrap
    user: kubelet-bootstrap
  name: bootstrap
current-context: bootstrap
kind: Config
preferences: {}
users:
- name: kubelet-bootstrap
  user:
    token: c2ihj5.l6a8nj8rdxplezs9

user@ubuntu:~/metrics-server$
```

To ensure the kubelet requests a new signed serving certificate, we need to adjust the kubelet arguments to use server certificate rotation and force the cluster to distribute a new certificate to the kubelet. Add `--rotate-server-certificates` to your kubelet. For kubeadm-bootstrapped clusters, you can add it to the `kubeadm-flags.env` file:

```
user@ubuntu:~/metrics-server$ sudo nano /var/lib/kubelet/kubeadm-flags.env

user@ubuntu:~/metrics-server$ sudo cat /var/lib/kubelet/kubeadm-flags.env

KUBELET_KUBEADM_ARGS="--cgroup-driver=cgroupfs --network-plugin=cni --pod-infra-container-
image=k8s.gcr.io/pause:3.1 --rotate-server-certificates"

user@ubuntu:~/metrics-server$
```

> N.B. If your cluster does not have a kubeadm-flags.env file, then edit the kubelet systemd service file directly and place `--rotate-server-certificates` inline with the ExecStart line pointing to the kubelet binary.

Next, we need to force the kubelet to use the bootstrap configuration by removing the existing kubelet.conf file. Renaming the old file will suffice:

```
user@ubuntu:~/metrics-server$ sudo mv /etc/kubernetes/kubelet.conf /etc/kubernetes/kubelet.old
```

Now restart the kubelet:

```
user@ubuntu:~/metrics-server$ sudo systemctl restart kubelet

user@ubuntu:~/metrics-server$ sudo systemctl status kubelet

● kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/lib/systemd/system/kubelet.service; enabled; vendor preset: enabled)
  Drop-In: /etc/systemd/system/kubelet.service.d
           └─10-kubeadm.conf
   Active: active (running) since Fri 2020-01-24 16:19:01 PST; 2s ago
     Docs: https://kubernetes.io/docs/home/
 Main PID: 125944 (kubelet)
    Tasks: 15
   Memory: 34.5M
      CPU: 420ms
   CGroup: /system.slice/kubelet.service
           └─125944 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-
kubelet.conf --kubeconfig=/etc/kubernetes/kubelet.conf --config=/var/lib/kubelet/config.yaml --
cgroup-driver=cgroupfs

Jan 24 16:19:02 ubuntu kubelet[125944]: I0124 16:19:02.434257  125944 reconciler.go:207]
operationExecutor.VerifyControllerAttachedVolume started for volume "etcd-certs" (UniqueName:
"kubernetes.io/host-
Jan 24 16:19:02 ubuntu kubelet[125944]: I0124 16:19:02.434268  125944 reconciler.go:207]
operationExecutor.VerifyControllerAttachedVolume started for volume "weavedb" (UniqueName:
"kubernetes.io/host-pat
Jan 24 16:19:02 ubuntu kubelet[125944]: I0124 16:19:02.434282  125944 reconciler.go:207]
operationExecutor.VerifyControllerAttachedVolume started for volume "dbus" (UniqueName:
"kubernetes.io/host-path/d
Jan 24 16:19:02 ubuntu kubelet[125944]: I0124 16:19:02.434295  125944 reconciler.go:207]
operationExecutor.VerifyControllerAttachedVolume started for volume "config-volume" (UniqueName:
"kubernetes.io/co
Jan 24 16:19:02 ubuntu kubelet[125944]: I0124 16:19:02.434308  125944 reconciler.go:207]
operationExecutor.VerifyControllerAttachedVolume started for volume "config-volume" (UniqueName:
"kubernetes.io/co
Jan 24 16:19:02 ubuntu kubelet[125944]: I0124 16:19:02.434313  125944 reconciler.go:154]
Reconciler: start to sync state

...

q

user@ubuntu:~/metrics-server$
```

The kubelet will restart and use the bootstrap configuration since you removed the kubelet.conf. The bootstrap configuration file location is declared in the kubelet service's `--bootstrap-kubeconfig` argument, and the permanent kubeconfig file is declared using the `--kubeconfig` argument.

Check if a new certificate signing request for the kubelet is present. This is the kubelet's request for a serving certificate:

```
user@ubuntu:~/metrics-server$ kubectl get csr

NAME        AGE    REQUESTOR                 CONDITION
csr-2wnsr   2m6s   system:bootstrap:c2ihj5   Approved,Issued
csr-485fp   46s    system:node:ubuntu        Pending

user@ubuntu:~/metrics-server$
```

Use `kubectl certificate approve` on the Pending node certificate:

```
user@ubuntu:~/metrics-server$ kubectl certificate approve csr-485fp

certificatesigningrequest.certificates.k8s.io/csr-485fp approved

user@ubuntu:~/metrics-server$ kubectl get csr

NAME        AGE     REQUESTOR                 CONDITION
csr-2wnsr   2m42s   system:bootstrap:c2ihj5   Approved,Issued
csr-485fp   82s     system:node:ubuntu        Approved,Issued

user@ubuntu:~/metrics-server$
```

The kubelet certificate is now running with a certificate signed by the cluster CA, meaning the API server and other services within the cluster will successfully validate any requests. This process needs to be performed for all kubelets.

3b. Reconfigure the metrics server to use the internal IP addresses of the kubelets

Another issue we need to address is how the Metrics Server communicates with the kubelets. The default behavior of the metrics server is to try and connect to the kubelets via their hostnames. Display the local node status and see how the node presents itself to the cluster:

```
user@ubuntu:~/metrics-server$ kubectl describe $(kubectl get node -o name) | grep -A2 Addresses

Addresses:
  InternalIP:  192.168.229.132
  Hostname:    ubuntu

user@ubuntu:~/metrics-server$
```

Notice that the Node has a few IP address options and a hostname. Given that the metrics server runs in a Pod and may not have the ability to look up the hostname of the Kubelet's machine, so using the hostname (the metrics server default) is not ideal.

We can fix this problem by editing the Metrics Server deployment to pass the following argument to the Metrics Server:

- `--kubelet-preferred-address-types=InternalIP` - This will tell the Metrics Server to try to communicate using the Kubelet's InternalIP.

Display the deployments in the kube-system namespace:

```
user@ubuntu:~/metrics-server$ kubectl get deploy -n kube-system

NAME             READY   UP-TO-DATE   AVAILABLE   AGE
coredns          2/2     2            2           24h
metrics-server   1/1     1            1           32m

user@ubuntu:~/metrics-server$
```

Make the changes to the Metrics Server deployment:

```
user@ubuntu:~/metrics-server$ kubectl edit -n kube-system deploy metrics-server

...
```

In the editor add the `args` key and switches shown below:

```
...

template:
  metadata:
    creationTimestamp: null
    labels:
      k8s-app: metrics-server
    name: metrics-server
  spec:
    containers:
    - args:
      - --kubelet-preferred-address-types=InternalIP
      - --cert-dir=/tmp
      - --secure-port=4443
      image: k8s.gcr.io/metrics-server-amd64:v0.3.6
      imagePullPolicy: Always
      name: metrics-server

...
```

Save your changes and exit:

```
...

deployment.apps/metrics-server edited

user@ubuntu:~/metrics-server$
```

Checking the pods again, you will see that the metrics server pod has been recreated and shows an age in seconds:

```
user@ubuntu:~/metrics-server$ kubectl get po -n kube-system

NAME                              READY   STATUS    RESTARTS   AGE
coredns-5644d7b6d9-gd8tv          1/1     Running   0          24h
coredns-5644d7b6d9-p6rd6          1/1     Running   0          24h
etcd-ubuntu                       1/1     Running   0          24h
kube-apiserver-ubuntu             1/1     Running   0          24h
kube-controller-manager-ubuntu    1/1     Running   0          24h
kube-proxy-4c8tb                  1/1     Running   0          24h
kube-scheduler-ubuntu             1/1     Running   0          24h
metrics-server-c84799697-mhmwq    1/1     Running   0          36s
weave-net-76twd                   2/2     Running   0          24h

user@ubuntu:~/metrics-server$
```

It will take two or three minutes for the new Metrics Server to collect data but if you try a few times you should now see that top returns actual metrics. Use the Linux `watch` (not to be confused with `kubectl -watch`) tool to continually send a kubectl command. Use CTRL C once you see that the metrics are being reported:

```
user@ubuntu:~/metrics-server$ watch kubectl top node

...
```

```
Every 2.0s: kubectl top node                          Fri Jan 24 16:23:47 2020

NAME     CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
ubuntu   140m         7%     1995Mi          52%


^C

user@ubuntu:~/metrics-server$
```

Try `kubectl top` on pods:

```
user@ubuntu:~/metrics-server$ kubectl top pod -n kube-system

NAME                              CPU(cores)   MEMORY(bytes)
coredns-5644d7b6d9-gd8tv          2m           15Mi
coredns-5644d7b6d9-p6rd6          2m           24Mi
etcd-ubuntu                       7m           59Mi
kube-apiserver-ubuntu             27m          301Mi
kube-controller-manager-ubuntu    6m           49Mi
kube-proxy-4c8tb                  1m           22Mi
kube-scheduler-ubuntu             1m           21Mi
metrics-server-c84799697-mhmwq    1m           12Mi
weave-net-76twd                   1m           94Mi

user@ubuntu:~/metrics-server$
```

Awesome, metrics up and running.

4. Autoscaling

Kubernetes 1.7 and later use the metrics-server to provide metrics used by the autoscaler controller. The base metrics used in most autoscaling scenarios are CPU and memory.

To test autoscaling we'll need:

- A target pod/container-image that we can load
- A pod we can use to drive the target pod
- A Horizontal Pod Autoscalor (HPA) to scale the target pod when it gets busy

To begin, create a simple php web server we can use as our target:

```
user@ubuntu:~/metrics-server$ mkdir ~/hpa && cd ~/hpa

user@ubuntu:~/hpa$ nano index.php && cat index.php
```

```
<?php
  $x = 0.0001;
  for ($i = 0; $i <= 1000000; $i++) {
    $x += sqrt($x);
  }
  echo "OK!";
?>
```

```
user@ubuntu:~/hpa$
```

This app will simply compute a million square roots each time it is hit.

Now create a Dockerfile to create a container image for the PHP server:

```
user@ubuntu:~/hpa$ nano Dockerfile && cat Dockerfile

FROM php:5-apache
ADD index.php /var/www/html/index.php
RUN chmod a+rx /var/www/html/index.php

user@ubuntu:~/hpa$
```

Build a container image for the PHP server:

```
user@ubuntu:~/hpa$ docker image build -t target .

Sending build context to Docker daemon  3.072kB
Step 1/3 : FROM php:5-apache
5-apache: Pulling from library/php
5e6ec7f28fb7: Pull complete
cf165947b5b7: Pull complete
7bd37682846d: Pull complete
99daf8e838e1: Pull complete
ae320713efba: Pull complete
ebcb99c48d8c: Pull complete
9867e71b4ab6: Pull complete
936eb418164a: Pull complete
bc298e7adaf7: Pull complete
ccd61b587bcd: Pull complete
b2d4b347f67c: Pull complete
56e9dde34152: Pull complete
9ad99b17eb78: Pull complete
Digest: sha256:0a40fd273961b99d8afe69a61a68c73c04bc0caa9de384d3b2dd9e7986eec86d
Status: Downloaded newer image for php:5-apache
 ---> 24c791995c1e
Step 2/3 : ADD index.php /var/www/html/index.php
 ---> 170b6d61f08f
Step 3/3 : RUN chmod a+rx /var/www/html/index.php
 ---> Running in 105f9f2ee939
Removing intermediate container 105f9f2ee939
 ---> ddafae5353b4
Successfully built ddafae5353b4
Successfully tagged target:latest

user@ubuntu:~/hpa$
```

> N.B. Remember that Docker build places the PHP image on the node that you built it on! If you are running a multi-node Kubernetes cluster, make sure that any pods using this image run on a node where the image is present.

Now we can launch a standard single replica deployment with our new server:

```
user@ubuntu:~/hpa$ kubectl create deploy web1 --image=target --dry-run -o yaml > target-
deploy.yaml

user@ubuntu:~/hpa$ nano target-deploy.yaml

user@ubuntu:~/hpa$ cat target-deploy.yaml

apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: web1
  name: web1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: web1
```

```
  template:
    metadata:
      labels:
        app: web1
    spec:
      nodeName: ubuntu
      containers:
      - image: target
        imagePullPolicy: Never
        name: target
        resources:
          requests:
            cpu: "200m"

user@ubuntu:~/hpa$ kubectl create -f target-deploy.yaml

deployment.apps/web1 created

user@ubuntu:~/hpa$ kubectl expose deploy web1 --port 80

service/web1 exposed

user@ubuntu:~/hpa$
```

Our pod will request 200 mils of CPU (1/5 of a CPU) and we also set `imagePullPolicy` to "Never" to prevent the kubelet from asking Docker to download the image (which it would not find since we have just built it locally).

> N.B. A nodeName key keeps this deployment's pods from running on a node that doesn't have the PHP image you built. Make sure to set it to the hostname of the node that has the PHP image.

List your resources:

```
user@ubuntu:~/hpa$ kubectl get deploy,rs,po,svc

NAME                   READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/web1   1/1     1            1           47s

NAME                              DESIRED   CURRENT   READY   AGE
replicaset.apps/web1-75c54b6579   1         1         1       47s

NAME                         READY   STATUS    RESTARTS   AGE
pod/web1-75c54b6579-lqf7z    1/1     Running   0          47s

NAME                 TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP   24h
service/web1         ClusterIP   10.103.152.45   <none>        80/TCP    44s

user@ubuntu:~/hpa$
```

So we now have our deployment with a replica set, one pod and a service (created because of the `--expose` switch).

Now let's create an HPA for our deployment:

```
user@ubuntu:~/hpa$ kubectl autoscale deployment web1 --cpu-percent=50 --min=1 --max=5

horizontalpodautoscaler.autoscaling/web1 autoscaled

user@ubuntu:~/hpa$
```

Our HPA will now scale up when our set of pods exceed 50% of their desired CPU resources, in this case 100 mils. Right now we have one pod with a request for 20% of a CPU, or 200 mils. Display the HPA. You may see the Targets for the pod read for a few minutes until the autoscaler communicates with the metrics server:

```
user@ubuntu:~/hpa$ kubectl get hpa
```

```
NAME   REFERENCE         TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
web1   Deployment/web1   0%/50%    1         5         1          63s

user@ubuntu:~/hpa$
```

Now display the resources used by your pod:

```
user@ubuntu:~/hpa$ kubectl top pod

NAME                    CPU(cores)   MEMORY(bytes)
web1-75c54b6579-lqf7z   1m           8Mi

user@ubuntu:~/hpa$
```

Our pod is using 1 mil of its 200 mils. It will need to get to 100 mils before the HPA will add another pod to the deployment. Let's create some load!

Run an interactive busybox pod to hit the PHP server from:

```
user@ubuntu:~/hpa$ kubectl run --generator=run-pod/v1 -it driver --image=busybox

If you don't see a command prompt, try pressing enter.

/ #
```

Now start a loop that will make continuous requests of our 1mm square root service:

```
/ # while true; do wget -q -O- http://web1.default.svc.cluster.local; done
OK!
OK!
OK!

...
```

While the requests are running, **open a new terminal** and check the pod utilization with top:

```
user@ubuntu:~$ kubectl top pod

NAME                    CPU(cores)   MEMORY(bytes)
web1-75c54b6579-lqf7z   1m           8Mi

user@ubuntu:~$ kubectl top pod

NAME                    CPU(cores)   MEMORY(bytes)
driver                  7m           1Mi
web1-75c54b6579-lqf7z   928m         10Mi

user@ubuntu:~$
```

Now over 900 mils, that is well over the threshold of 100 mils, why don't we have more pods reporting metrics? The HPA will not respond to spikes because this would trash the cluster, creating and deleting pods wastefully. Rather the HPA waits for a configurable period (defined by Controller Manager settings, defaulting to about 3 minutes) and then begins to scale.

Run top several more times until you see scaling activity, display the HPA status now and then also:

```
user@ubuntu:~$ kubectl top pod

NAME                    CPU(cores)   MEMORY(bytes)
driver                  7m           1Mi
```

```
web1-75c54b6579-lqf7z   928m         10Mi

user@ubuntu:~$ kubectl get hpa

NAME   REFERENCE         TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
web1   Deployment/web1   83%/50%   1         5         5          3m11s

user@ubuntu:~$ kubectl top pod

NAME                    CPU(cores)   MEMORY(bytes)
driver                  10m          2Mi
web1-75c54b6579-gjss2   170m         10Mi
web1-75c54b6579-k5tqp   164m         10Mi
web1-75c54b6579-lqf7z   203m         10Mi
web1-75c54b6579-psm69   130m         10Mi
web1-75c54b6579-w6d4c   170m         10Mi

user@ubuntu:~$
```

You should see the HPA scale the pods out to the limit of 5 configured. Once those pods communicate with the metrics server, the start showing up on `kubectl top` output. Display your deployment:

```
user@ubuntu:~$ kubectl get deploy

NAME   READY   UP-TO-DATE   AVAILABLE   AGE
web1   5/5     5            5           4m32s

user@ubuntu:~$
```

Note that the only thing the HPA actually does is change the replica count on the deployment. The deployment and replica set do the rest.

Use `control C` to stop the driver loop and exit the pod:

```
...

OK!
OK!

^C

/ # exit

Session ended, resume using 'kubectl attach driver -c driver -i -t' command when the pod is
running

user@ubuntu:~/hpa$
```

Examine the metrics and HPA:

```
user@ubuntu:~/hpa$ kubectl get hpa

NAME   REFERENCE         TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
web1   Deployment/web1   90%/50%   1         5         5          3m58s

user@ubuntu:~/hpa$ kubectl top pod

NAME                    CPU(cores)   MEMORY(bytes)
driver                  10m          2Mi
web1-75c54b6579-gjss2   122m         10Mi
web1-75c54b6579-k5tqp   182m         10Mi
web1-75c54b6579-lqf7z   202m         10Mi
web1-75c54b6579-psm69   211m         10Mi
web1-75c54b6579-w6d4c   192m         10Mi
```

```
user@ubuntu:~/hpa$
```

The HPA will wait to ensure traffic does not return for a few minutes before scaling down. While you wait for the HPA to scale down, describe the resource:

```
user@ubuntu:~/hpa$ kubectl describe hpa web1

Name:                                                  web1
Namespace:                                             default
Labels:                                                <none>
Annotations:                                           <none>
CreationTimestamp:                                     Fri, 24 Jan 2020 16:28:17 -0800
Reference:                                             Deployment/web1
Metrics:                                               ( current / target )
  resource cpu on pods  (as a percentage of request):  0% (1m) / 50%
Min replicas:                                          1
Max replicas:                                          5
Deployment pods:                                       5 current / 5 desired
Conditions:
  Type            Status  Reason               Message
  ----            ------  ------               -------
  AbleToScale     True    ScaleDownStabilized  recent recommendations were higher than current
one, applying the highest recent recommendation
  ScalingActive   True    ValidMetricFound     the HPA was able to successfully calculate a
replica count from cpu resource utilization (percentage of request)
  ScalingLimited  True    TooManyReplicas      the desired replica count is more than the
maximum replica count
Events:
  Type     Reason                        Age    From                       Message
  ----     ------                        ----   ----                       -------
  Warning  FailedGetResourceMetric       4m28s  horizontal-pod-autoscaler  did not receive
metrics for any ready pods
  Warning  FailedComputeMetricsReplicas  4m28s  horizontal-pod-autoscaler  invalid metrics (1
invalid out of 1), first error is: failed to get cpu utilization: did not receive metrics for
any ready pods
  Normal   SuccessfulRescale             3m12s  horizontal-pod-autoscaler  New size: 4; reason:
cpu resource utilization (percentage of request) above target
  Normal   SuccessfulRescale             2m57s  horizontal-pod-autoscaler  New size: 5; reason:
cpu resource utilization (percentage of request) above target

user@ubuntu:~/hpa$
```

Notice the scaling events are reported at the bottom of the description.

The HPA abides by a setting called `horizontal-pod-autoscaler-downscale-stabilization` which is passed as a flag to the cluster's kube-controller-manager. This argument prevents an HPA from scaling down for 5 minutes by default. After 5 minutes, the HPA will scale the deployment back down to one pod.

Use `watch` to repeatedly run the `kubectl get hpa` and `kubectl top pod` commands:

```
user@ubuntu:~/hpa$ watch 'kubectl get hpa && kubectl top pod'

Every 2.0s: kubectl get hpa && kubectl top pod          Fri Jan 24 16:33:44 2020

NAME   REFERENCE         TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
web1   Deployment/web1   0%/50%    1         5         5          5m27s
NAME                     CPU(cores)   MEMORY(bytes)
driver                   0m           1Mi
web1-75c54b6579-gjss2    1m           10Mi
web1-75c54b6579-k5tqp    1m           10Mi
web1-75c54b6579-lqf7z    1m           10Mi
web1-75c54b6579-psm69    1m           10Mi
web1-75c54b6579-w6d4c    1m           10Mi


user@ubuntu:~/hpa$
```

After about five minutes, you will see the deployment scale down. Use `Ctrl c` to exit the watch when you see the scale down event:

```
Every 2.0s: kubectl get hpa && kubectl top pod          Fri Jan 24 16:37:47 2020

NAME   REFERENCE         TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
web1   Deployment/web1   0%/50%    1         5         5          9m30s
NAME                     CPU(cores)   MEMORY(bytes)
driver                   0m           1Mi
web1-75c54b6579-lqf7z    1m           10Mi

^C

user@ubuntu:~/hpa$
```

Describe the HPA once more and you will see a scaling event at the bottom that reports a new size and the reason:

```
user@ubuntu:~/hpa$ kubectl describe hpa | grep Events -A10

Events:
  Type     Reason                        Age    From                       Message
  ----     ------                        ----   ----                       -------
  Warning  FailedGetResourceMetric       9m50s  horizontal-pod-autoscaler  did not receive
metrics for any ready pods
  Warning  FailedComputeMetricsReplicas  9m50s  horizontal-pod-autoscaler  invalid metrics (1
invalid out of 1), first error is: failed to get cpu utilization: did not receive metrics for
any ready pods
  Normal   SuccessfulRescale             8m34s  horizontal-pod-autoscaler  New size: 4; reason:
cpu resource utilization (percentage of request) above target
  Normal   SuccessfulRescale             8m19s  horizontal-pod-autoscaler  New size: 5; reason:
cpu resource utilization (percentage of request) above target
  Normal   SuccessfulRescale             46s    horizontal-pod-autoscaler  New size: 1; reason:
All metrics below target

user@ubuntu:~/hpa$
```

Your cluster is now capable of dynamically sizing its elements based on metrics measured from incoming workloads.

5. Clean up

List all of your resources:

```
user@ubuntu:~/hpa$ kubectl get all -o=name

pod/driver
pod/web1-75c54b6579-lqf7z
service/kubernetes
service/web1
deployment.apps/web1
replicaset.apps/web1-75c54b6579
horizontalpodautoscaler.autoscaling/web1

user@ubuntu:~/hpa$
```

Now delete everything you created for this lab (but don't delete the "kubernetes" service!!):

```
user@ubuntu:~/hpa$ kubectl delete service/web1 deploy/web1 pod/driver hpa/web1

service "web1" deleted
deployment.apps "web1" deleted
pod "driver" deleted
horizontalpodautoscaler.autoscaling "web1" deleted
```

```
user@ubuntu:~/hpa$
```

Congratulations, you have completed the lab!