THE NEW PaaS:

Using Docker and Containers to Simplify Your Life and Accelerate Development on AWS

LAB 2: KUBERNETES

Version 1.2

Table of Contents

Introduction

Overview

In this lab we will explore the basic components of a Kubernetes cluster. First we will have a look at the `etcd` key store service and then we will start building up a simple guestbook application[1] in a single-host Kubernetes environment.

What is Kubernetes?

Kubernetes is an open-source project started by Google, with the design goal of being a cluster management tool for hosts systems running Docker. You can learn more here:

https://github.com/GoogleCloudPlatform/kubernetes

Technical Knowledge Prerequisites

To successfully complete this lab, you should be familiar with SSH or PuTTY, with the Linux command-line environment, and with Docker. **For an introduction to Docker, refer to Lab 1 of this series.**

Topics Covered

This lab will take you through Kubernetes configuration and operation, including:

- `etcd`, the cluster configuration backplane
- Working with Kubernetes concepts like pods, services and replicationControllers
- Deploying a basic multi-component system in a Kubernetes cluster

Sign in to the AWS Management Console

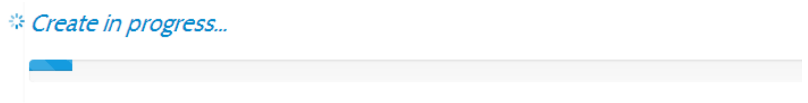
If you are taking the entire three-lab course, you should still be logged in to the AWS Console from **Lab 1: Docker**. If so, you can skip this section.

Using qwikLABS^{tm} to sign in to the AWS Management Console

Welcome to this self-paced lab! The first step is for you to sign in to Amazon Web Services.

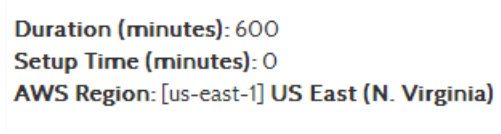
1. To the right of the lab title, click **Start Lab**. If you are prompted for a token, use the one you received or purchased.

   Note: A status bar shows the progress of the lab environment creation process. The AWS Management Console is accessible during lab resource creation, but your AWS resources may not be fully available until the process is complete.



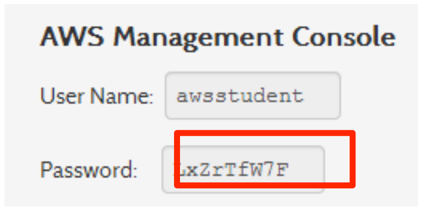
[1] This exercise is based on example code from the Kubernetes code base: https://github.com/GoogleCloudPlatform/kubernetes/tree/master/examples/guestbook

2. On the lab details page, notice the lab properties.
   a. **Duration -** The time the lab will run before automatically shutting down.
   b. **Setup Time -** The estimated time to set up the lab environment.
   c. **AWS Region** - The AWS Region in which the lab resources are created.

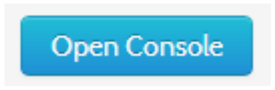

Duration (minutes): 600
Setup Time (minutes): 0
AWS Region: [us-east-1] US East (N. Virginia)

Note: The AWS Region for your lab will differ depending on your location and the lab setup.

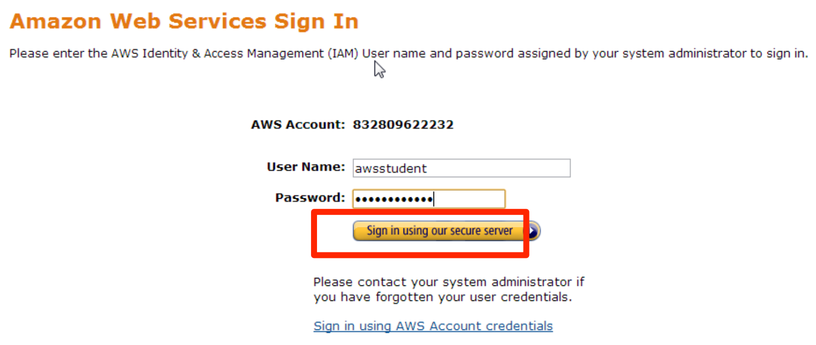
3. In the AWS Management Console section of the qwikLAB™ page, copy the Password to the clipboard.


AWS Management Console
User Name: awsstudent
Password: ;xZrTfW7F

4. Click the Open Console button.



5. Log into the AWS Management Console using the following steps.
   a. In the **User Name** field type **awsstudent**.
   b. In the **Password** field, paste the password copied from the lab details page.
   c. Click **Sign in using our secure server**.


Amazon Web Services Sign In
Please enter the AWS Identity & Access Management (IAM) User name and password assigned by your system administrator to sign in.
AWS Account: 832809622232
User Name: awsstudent
Password: ••••••••••••

Please contact your system administrator if you have forgotten your user credentials.
Sign in using AWS Account credentials

Note: The AWS account is automatically generated by *qwikLAB*™. Also, the login credentials for the awsstudent account are provisioned by *qwikLAB*™ using AWS Identity Access Management.

SSH Access

The **Lab 1: Docker** guide contains detailed information on how to establish an SSH connection to an EC2 instance using the `aws-reinvent-2014` key. Refer there for information on how to connect to the `lab2_kubernetes` instance.

Module 1: `etcd`

In this section we will have a look at the `etcd` key/value store, which serves as the configuration backplane of the Kubernetes cluster.

Once you have logged in to the Kubernetes host, you can see that there is already a running Docker container. Recall from the previous lab that you can see what's happening in Docker with `docker ps`:

```
$ docker ps
```

This output has been trimmed down, but you should notice the following values:

```
IMAGE                 PORTS                                         NAMES
nhripps/etcd:latest   0.0.0.0:4001->4001/tcp, 0.0.0.0:7001->7001/tcp  etcd
```

This tells us a few things about the current state of this system:

- It is running the `etcd` data store from a docker container
- `etcd`'s internal ports have been mapped to same-number ports on the host

Kubernetes is capable of running `etcd` outside of a Docker container, but for the purposes of this lab we are running it separately from the rest of Kubernetes.

1. Creating a key/value setting

The primary interface to `etcd` is a RESTful web API. So, we can create a new key/value pair on the instance with a curl command, like so:

```
curl -L -X PUT http://127.0.0.1:4001/v2/keys/message -d value="Hello"
```

The output should look similar to this; it is formatted here for easier readability:

```
{ "action":"set",
  "node":{
    "key":"/message",
    "value":"Hello",
    "modifiedIndex":5,
    "createdIndex":5
  }
}
```

2. Getting the value of an existing key

If you are familiar with RESTful APIs then you can probably guess what happens now. The HTTP PUT method enables clients to create key/value pairs, and the HTTP GET method enables clients to retrieve values:

```
curl -L -X GET http://127.0.0.1:4001/v2/keys/message
```

The output should be the same bare JSON that was returned when they key/value pair was created.

3. Deleting a key/value pair

Finally, the HTTP DELETE method enables us to remove a key that is no longer in use:

```
curl -L -X DELETE http://127.0.0.1:4001/v2/keys/message
```

Any subsequent attempts to retrieve the message key will result in an error response from `etcd`.

4. Convenience utility `etcdctl`

As a convenience, the `etcdctl` command-line tool[2] supports the full range of functionality offered by `etcd`'s API:

```
$ etcdctl set foo 'bar'
bar

$ ectdctl ls /
/foo

$ etcdctl get foo
bar

$ etcdctl rm foo
```

[2] Source code available here: https://github.com/coreos/etcdctl

EXERCISE

You can try some other operations with `etcdctl` as well:

- Use `etcdctl mkdir` to create a directory structure.
- Monitor a key with `etcdctl watch` in one terminal, and then update the key with `etcdctl set` in another.

Feeling comfortable with the way `etcd` enables systems to easily get and set key/value pairs? Nice work! Head on to the next section.

Module 2: Kubernetes

Now that we've kicked the tires on `etcd` a bit, it is time to put Kubernetes to work. For the sake of simplicity, we will run the Kubernetes stack as a foreground process in one terminal, and perform operations with `kubecfg` in another terminal.

1. Get the system running

Open a second terminal and log in to the Kubernetes host. Now, back in the first terminal, run this command to initiate the Kubernetes services:

```
/opt/kubernetes/hack/local-up-cluster.sh
```

You should see the following output:

```
$ /opt/kubernetes/hack/local-up-cluster.sh
Building local go components
Local Kubernetes cluster is running. Press Ctrl-C to shut it
down.
Logs:
  /tmp/apiserver.log
  /tmp/controller-manager.log
  /tmp/kubelet.log
  /tmp/kube-proxy.log
  /tmp/k8s-scheduler.log
```

What just happened? Jumping to the second window, you can have a look at the contents of the local-up-cluster.sh script:

```
$ less /opt/kubernetes/hack/local-up-cluster.sh
```

In summary, here's what it does:

- *It starts the Kubernetes API server.*
  This is the RESTful API that will allow us to work with pods, services and replicationControllers. As we will see further on, it has a lot of similarities with etcd's own JSON-based web API.
- *It starts the controller-manager.*
  This service compares the replicationController definitions to the running pod info reported by each minion. Not enough instances of a pod? Too many instances? In either of these cases, controller-manager will tell the minions to start or stop pods as appropriate.

- *It starts kubelet.*
  Kubelet is the minion's Kubernetes agent. This is the daemon that watches for pod and replicationController changes via etcd and then sends instructions to the local Docker instance accordingly.
- *It starts kube-proxy.*
  Kube-proxy also runs on minions. It does for Kubernetes services what kubelet does for pods and replicationControllers: it listens for changes on etcd and then reconfigures port-proxying on the minion accordingly.
- *Finally, this script starts the scheduler.*
  This service is part of the Kubernetes master; it takes incoming instructions and decides which minions will be assigned with the new tasks.

In your second terminal, you can verify that these processes are running with the ps command:

```
$ ps -elf | grep kubernetes
```

2. Create a pod for our datastore master

The first step in setting up our guestbook is to establish a single Redis master instance. Since there will only be one in our deployment, we can define it as a simple single-image pod.

In the `/root/guestbook` directory of your Kubernetes host, have a look at the **`redis-master.json`** file.

```
{
  "id": "redis-master-2",
  "kind": "Pod",
  "apiVersion": "v1beta1",
  "desiredState": {
    "manifest": {
      "version": "v1beta1",
      "id": "redis-master-2",
      "containers": [{
        "name": "master",
        "image": "nhripps/redis",
        "ports": [{
          "containerPort": 6379,
          "hostPort": 6379
        }]
      }]
    }
  },
  "labels": {
    "name": "redis-master"
  }
}
```

A few of the fields are called out in boldface:

- **containers**:
  A pod consists of one or more docker containers that will all be managed together on a single minion. Each record indicates the source Docker image, the container-to-host port mappings, and the name for the running container.

- **labels**:
  A pod can have zero or more labels, each of which consists of a key and a value. The only restriction is that the key portion of each label must be unique for that pod (meaning that a pod can't have two "name=*" labels, for instance). Labels are used by Kubernetes to select matching pods across all of the minions in a cluster.

To add this pod to the cluster, we run the `kubecfg` utility:

```
$ kubecfg -c redis-master.json create pods
```

And the output will look like this:

```
ID              Image(s)          Host         Labels              Status
----------      ----------        ----------   ----------          -------
redis-master-2  dockerfile/redis  /            name=redis-master   Waiting
```

We can gain some insight into what the `create pods` command is doing by looking at the Docker side of things with `docker ps`. It may take a few minutes, but when the Kubernetes scheduler starts the pod, you will see a Docker container based on the dockerfile/redis image:

```
$ docker ps
CONTAINER ID        IMAGE                      COMMAND
33d3cfa9f579        nhripps/redis:latest       redis-server /etc/re
3ad9cb54ae98        kubernetes/pause:latest    /pause
49bfc12ff68c        nhripps/etcd:latest        /opt/etcd/bin/etcd /
```

Once you see that, you can also see a change of state in Kubernetes by listing the currently registered pods:

```
# kubecfg list pods
ID                  Image(s)            Host                Labels              Status
----------          ----------          ----------          ----------          -------
redis-master-2      dockerfile/redis    127.0.0.1/          name=redis-master   Running
```

While we're talking about the `list` function, note that you can also list minions, services, and replicationControllers. Right now there isn't much to see, but there will be soon…

EXERCISE

Using the `redis-master.json` file as a template, define and load a pod of your own based on any Docker image. Watch the Kubernetes and Docker log files to see the progress as the image is downloaded and deployed.

3. Create a service for the datastore master

We have a running Redis master instance, but right now there is no way for the cluster to route network traffic to that container. It is time to define a service. Have a look at the `redis-master-service.json` file:

```
{
  "id": "redismaster",
  "kind": "Service",
  "apiVersion": "v1beta1",
  "port": 10000,
  "containerPort": 6379,
  "selector": {
    "name": "redis-master"
  }
}
```

The `port` and `containerPort` values are instructions to the `kube-proxy` service. Every minion in the cluster will route traffic coming in on port 10000 to minions that are running the right pod. How does a minion know if it is running the correct pod? The `selector` object contains all of the pod labels that identify eligible pods.

In this case, the pod that we just created includes the `name=redis-master` label and so network traffic on port 10000 of our minion will be routed to that container.

To instantiate this service, we invoke `kubecfg`:

```
$ kubecfg -c redis-master-service.json create services
ID                  Labels              Selector            Port
----------          ----------          ----------          ----------
redismaster                             name=redis-master   10000
```

EXERCISE

Using the `redis-master-service.json` file as a template, create another service. If you set up a pod during the previous exercise, have this new service map to that pod's `containerPort`.

4. Turn up several datastore slave pods at once with a replicationController

Now we that we have our datastore master, we can achieve some fault-tolerance with a number of datastore slave pods. Defining them as a replicationController enables us to manage them as a group and also causes the controller-manager service to maintain our desired number of running instances.

The `redis-slave-controller.json` file contains the configuration information for this replicationController:

```
{
  "id": "redisSlaveController",
  "kind": "ReplicationController",
  "apiVersion": "v1beta1",
  "desiredState": {
    "replicas": 2,
    "replicaSelector": {"name": "redisslave"},
    "podTemplate": {
      "desiredState": {
        "manifest": {
          "version": "v1beta1",
          "id": "redisSlaveController",
          "containers": [{
            "name": "slave",
            "image": "nhripps/redis-slave",
            "ports": [{"containerPort": 6379, "hostPort": 6380}]
          }]
        }
      },
      "labels": {"name": "redisslave"}
  }},
  "labels": {"name": "redisslave"}
}
```

For this system, we will define a two-instance group, where instances are identified by the `name=redisslave` label. The podTemplate definition object we see here is structurally identical to the pod configuration that we saw in `redis-master.json`.

The `kubecfg` invocation is as follows:

```
$ kubecfg -c redis-slave-controller.json create replicationControllers
```

Now, because we are only running a single-instance cluster, the replicationController cannot fully instantiate. Both of the pods want to bind their container port 6379 to host port 6380. One succeeds, the other fails.

If you tail the `k8s-scheduler.log` file, you will see that the scheduler repeatedly attempts to start the second pod:

```
$ tail -f /tmp/k8s-scheduler.log
2106936a-3f3f-11e4-8950-22000b480fb3:{}]
I0918 14:31:07.976397 01608 factory.go:205] Attempting to bind
```

```
2106c148-3f3f-11e4-8950-22000b480fb3 to 127.0.0.1
I0918 14:31:08.009963 01608 request.go:292] Waiting for completion
of /operations/286
E0918 14:31:10.010825 01608 factory.go:131] Error scheduling
2106c148-3f3f-11e4-8950-22000b480fb3: Status: failure
(api.Status{JSONBase:api.JSONBase{Kind:"", ID:"",
CreationTimestamp:util.Time{Time:time.Time{sec:0, nsec:0x0,
loc:(*time.Location)(nil)}}, SelfLink:"", ResourceVersion:0x0,
APIVersion:""}, Status:"failure", Message:"The assignment would
cause a constraint violation", Reason:"",
Details:(*api.StatusDetails)(nil), Code:500}); retrying
I0918 14:31:10.010888 01608 factory.go:78] About to try and schedule
pod 2106c148-3f3f-11e4-8950-22000b480fb3
known minions: map[127.0.0.1:{}]
known scheduled pods: map[redis-master-2:{}]
```

Press `<CTRL>+C` to exit the tail.

You can modify the number of instances associated with a running replicationController using the `resize` command:

```
    $ kubecfg resize redisSlaveController 1
```

Doing so has the following chain effect:

- `controller-master` receives the new desired count and polls the cluster for the current actual count.
- On seeing that the current actual count is correct, `controller-master` cancels any outstanding requests for more pods of this type (selector: `name=redisslave`) in the scheduler queue.
- `k8s-scheduler` stops seeing orders for the redundant pod in its queue, has a soothing cup of tea[3], and gets on with its day.

When the cluster has sorted itself out, running docker ps should result in output like this:

```
$ docker ps
CONTAINER ID        IMAGE                              COMMAND
20f0846288bf        nhripps/redis-slave:latest         /bin/sh -c /run.sh
6516a5e39b4a        kubernetes/pause:latest            /pause
33d3cfa9f579        nhripps/redis:latest               redis-server /etc/re
```

[3] This part doesn't actually happen; `k8s-scheduler` is more of a coffee drinker. Also note that in the time since the lab was developed, a workaround for pod hostPort collisions has been added to the Kubernetes system.

```
3ad9cb54ae98        kubernetes/pause:latest           /pause
49bfc12ff68c        nhripps/etcd:latest               /opt/etcd/bin/etcd /
```

And the current list of pods will also reflect the replicationController member:

```
$ kubecfg list pods
ID                                      Image(s)
----------                              ----------
redis-master-2                          nhripps/redis
2106c148-3f3f-11e4-8950-22000b480fb3    nhripps/redis-slave
```

EXERCISE

Create a replicationController of your own using `redis-slave-controller.json` as a template. Make sure to specify a new `name` label for the `replicaSelector` and `labels` parameters. As with the pod exercise, you can choose any Docker image to replicate; if it isn't already on the host, Kubernetes will download it.

5. Create a service for the datastore slave replicationController

As we did with the redis-master-2 pod, we need to create kube-proxy rules for the new replicationController. Looking at redis-slave-service.json, we see that we continue to rely on pod labels to associate ports and containers:

```
{
  "id": "redisslave",
  "kind": "Service",
  "apiVersion": "v1beta1",
  "port": 10001,
  "containerPort": 6379,
  "labels": {
    "name": "redisslave"
  },
  "selector": {
    "name": "redisslave"
  }
}
```

In this instance, we will also apply a label to the service object itself. Create the new service configuration with kubecfg:

```
$ kubecfg -c redis-slave-service.json create services
```

Now we should have two defined services in total:

```
$ kubecfg list services
ID                  Labels              Selector            Port
----------          ----------          ----------          ----------
redismaster                             name=redis-master   10000
redisslave          name=redisslave     name=redisslave     10001
```

Just as we did with the controller-manager and k8s-scheduler, we can gain some insight about what the kube-proxy service is doing by tailing its log file:

```
# tail -f /tmp/kube-proxy.log
I0918 14:01:40.583733 01607 proxier.go:413] Adding a new service redismaster on TCP
port 10000
I0918 14:01:40.583837 01607 proxier.go:384] Listening for redismaster on
tcp:[::]:10000
I0918 14:01:50.376297 01607 config.go:128] Adding new endpoint from source api : [{{
redismaster 0001-01-01 00:00:00 +0000 UTC  15 } [172.17.0.3:6379]}]
I0918 14:01:50.376372 01607 roundrobin.go:104] LoadBalancerRR: Setting endpoints for
redismaster to [172.17.0.3:6379]
I0918 15:10:49.646904 01607 config.go:223] Adding new service from source api : [{{
redisslave 2014-09-18 15:10:49 +0000 UTC  586 } 10001 TCP map[name:redisslave]
map[name:redisslave] false {0 6379 }}]
I0918 15:10:49.646978 01607 proxier.go:398] Received update notice: [{JSONBase:{Kind:
ID:redismaster CreationTimestamp:2014-09-18 14:01:40 +0000 UTC SelfLink:
ResourceVersion:14 APIVersion:} Port:10000 Protocol:TCP Labels:map[]
Selector:map[name:redis-master] CreateExternalLoadBalancer:false ContainerPort:{Kind:0
IntVal:6379 StrVal:}} {JSONBase:{Kind: ID:redisslave CreationTimestamp:2014-09-18
15:10:49 +0000 UTC SelfLink: ResourceVersion:586 APIVersion:} Port:10001 Protocol:TCP
Labels:map[name:redisslave] Selector:map[name:redisslave]
CreateExternalLoadBalancer:false ContainerPort:{Kind:0 IntVal:6379 StrVal:}}]
I0918 15:10:49.647029 01607 proxier.go:413] Adding a new service redisslave on TCP
port 10001
I0918 15:10:49.647105 01607 proxier.go:384] Listening for redisslave on tcp:[::]:10001
I0918 15:10:55.948732 01607 config.go:128] Adding new endpoint from source api : [{{
redisslave 0001-01-01 00:00:00 +0000 UTC  587 } [172.17.0.5:6379]}]
I0918 15:10:55.948802 01607 roundrobin.go:104] LoadBalancerRR: Setting endpoints for
redisslave to [172.17.0.5:6379]
```

Notably, the algorithm in use for proxy-based load balancing is a simple round-robin. This is a pluggable feature of the Kubernetes system and other algorithms are likely to become available.

6. Create a front-end pod via replicationController

Even though we are only planning to deploy a single front-end pod for our guestbook, the benefits of deploying pods in the context of a replicationController make it the preferred method of doing so. If we add more minions to our cluster in the future, we can easily increase the number of front-end pods to take advantage of them using the kubecfg resize command.

Fire up the front-end with:

```
$ kubecfg -c frontend-controller.json create replicationControllers
```

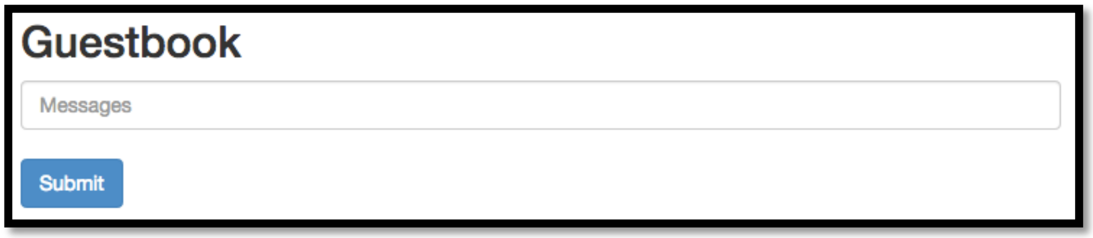
As with the previous steps, you can follow the progress of pod instantiation with docker ps, kubecfg list pods, and by tailing the Kubernetes services.

You should now be able to connect to the guestbook in your web browser by navigating to:

```
http://<kubernetes_host_ip>:8000/
```

You can try the helper function to quickly generate this link by running `lab_app_url` from the Kubernetes host terminal. If that doesn't work, you will need to look up the public IP of your Lab 2 host instance.

The guestbook should look like this:



Any time you enter some text and click the Submit button, it should appear below the Submit button on the page.

Congratulations! At this point you are running a complete end-to-end web service on Kubernetes and Docker!

Module 3: Extra Credit

Time permitting, here are a few more things to try:

Check out the Kubernetes structures in etcd

Using the etcdctl utility, you can see how Kubernetes uses the etcd service to orchestrate the cluster. Start by listing out the contents of the root level of the etcd keystore:

```
$ etcdctl ls /
```

Try the shorthand method for creating a replicationController

Because replicationControllers are the preferred method of instantiating pods, the kubecfg command supports a shorthand method for defining and deploying them in a single step.

The format of this usage is:

```
$ kubecfg [OPTIONS] [-p <port spec>] run <image> <replicas> <controller>
```

- `-p` - Specify container-to-host post mappings
- `image` - The name of the Docker image to use
- `replicas` - The number of pod instances to run
- `controller` - The unique name of this replicationController

Conclusion

Congratulations! You now have successfully:

- Seen how `etcd` can be used to set and get arbitrary key/value pairs
- Explored the main Kubernetes objects: pods, services and replicationControllers
- Created a complete web application by using the Kubernetes cluster to deploy associated Docker images

NOTE: If you will be continuing on to **Lab 3: OpenShift**, then *do not log out* of your AWS Console!

Additional Resources

For more information about Docker, Kubernetes and OpenShift, go to the OpenShift Origin repo on GitHub.