LFS258

Kubernetes Fundamentals

Version 1.0

LFS258: Version 1.0

Published by:

the **Linux Foundation**
http://www.linuxfoundation.org

LAB 2

Objective: Read parts of the canonical paper describing the roots of Kubernetes and join the Kubernetes community.

This first lab consists of a reading a technical paper, watching a video and joining the Kubernetes community.

- Read the Borg paper.
- Listen to John Wilkes talk about Borg and Kubernetes.
- Add the Kubernetes community hangout to your calendar, and attend at least once.
- Join the community on Slack and go in the **#kubernetes-users** channel.
- Check out the very active Stack Overflow community.

LAB 3

Objective: Review a useful Docker networking feature and start the basic components of the Kubernetes head node.

This lab is made of two activities:

- Use Docker networking to start containers that share the same network stack (i.e network namespace).
- Start a Kubernetes head node with **etcd**, the API server, and the controller manager.

In both activities, you need to be on a Docker host.

Docker networking

Docker allows you to specify different types of networking for your containers. One type that is very handy and used in Kubernetes is the *container* type. It allows you to share a network between two containers. In this exercise, you will illustrate this with three `bash` commands:

On a Docker host, start a `busybox` container that just waits. Then, start another container, using the `--net=container:<other_container_name>` option.

```
$ docker run -d --name=source busybox sleep 3600
$ docker run -d --name=same-ip --net=container:source busybox sleep 3600
```

Now, check the IP address of each container, something like the following should do:

```
$ docker exec -ti same-ip ifconfig
```

You should see that the second container inherited the same IP address as the *source* container. That is because, using the Docker `--net` option, we were able to use the same networking namespace to both containers.

That is what makes a Kubernetes **POD**, a set of containers running on the same host and that share the same networking namespace. This is what Kubernetes calls the *single IP-per-Pod model*: making a Pod almost look like a virtual machine running multiple processes with a single IP address.

Start your first Kubernetes head node

In this exercise, you will create the beginning of a Kubernetes cluster on your local Docker host. You will use Docker images to run **etcd**, the API server, and the controller manager.

In addition, those three components will share the same network, using the technique highlighted in the first exercise. This will show you how to run three processes in a “Pod”.

First, you need **etcd**, you can run it like this:

```
$ docker run -d --name=k8s -p 8080:8080 gcr.io/google_containers/etcd:2.2.1
etcd --data-dir /var/lib/data
```

Then, you will start the API server using the so-called `hyperkube` image, which contains the API server binary. We use a few simple settings to serve the API insecurely.

```
$ docker run -d --net=container:k8s
gcr.io/google_containers/hyperkube:v1.5.1 /apiserver
--etcd-servers=http://127.0.0.1:4001 \
--service-cluster-ip-range=10.0.0.1/24 \
--insecure-bind-address=0.0.0.0 \
--insecure-port=8080 \
--admission-control=AlwaysAdmit
```

Finally, you can start the `Admission` controller, which points to the API server.

```
$ docker run -d --net=container:k8s
gcr.io/google_containers/hyperkube:v1.5.1 /controller-manager
--master=127.0.0.1:8080
```

Notice that since **etcd**, the API server, and the controller manager share the same network namespace, they can reach each other on 127.0.0.1, even though they are running in different containers.

To test that, you have a working setup, use **etcdctl** in the **etcd** container, and list the what is the **`/registry`** directory:

```
$ docker exec -ti k8s etcdctl ls /registry
```

You can of course reach your Kubernetes API server and start exploring the API:

```
$ curl http://127.0.0.1:8080/api/v1
```

Note that in this exercise you did not start the scheduler, nor did you setup nodes with the **kubelet** and the **kube-proxy**. This is also a toy setup and you may want to set things up differently from scratch.

LAB 4

Objective: Using **`minikube`**, create a single node k8s cluster on your laptop. Use the **`kubectl`** CLI to validate that your installation works, and run the Ghost microblogging framework. This is illustrated in the screencast.

The remaining exercises in this course can all be done on **`minikube`**. In this exercise, we will install **minikube** and test that it is functioning properly. Below is the step-by-step walkthrough:

- Download **`minikube`** from the GitHub release page. Follow the instructions to do the installation.

- Download **`kubectl`** from the official release.

- Start the **`minikube`** VM and explore the various **`minikube`** commands:

  ```
  $ minikube start
  $ minikube --help
  ```

- Check the IP address of **`minikube`**

  ```
  $ minikube ip
  ```

- Check that you can reach the **`minikube`** instance with **`kubectl`**

  ```
  $ kubectl get nodes
  ```

  This should return **`minikube`**

- Run your first Pod

  ```
  $ kubectl run ghost --image=ghost
  ```


- Expose your application via a service

  `$ kubectl expose deployment/ghost --port=2368 --type=NodePort`

- Find the **`NodePort`** value of your service

  `$ kubectl get svc ghost -o yaml`

- Open your browser and check that Ghost is running.

Congratulations, at this stage you should have **`minikube`** running on your local machine and you will have deployed and accessed your first Kubernetes application. Do not worry if some of the concepts haven’t been explained yet, we will dive deeper in the next chapters.

LAB 5

Objective: Use **`minikube`**, explore the Kubernetes API, create your first Pod, create your first namespace, and start using the **`kubectl`** CLI

Explore the API

Start **`minikube`** and **`ssh`** to it:

```
$ minikube start
$ minikube ssh
```

You are now logged into **`minikube`**, you will see that you have Docker running and a few containers are already running. Access the API server and check the various endpoints:

```
$ curl localhost:8080
$ curl localhost:8080/version
$ curl localhost:8080/api/v1
```

You will see the various API groups, and the resources that belong to each group.

Create your first Pod

Exit **`minikube`** to get back on a terminal in your local machine, check that you have **`kubectl`** running and list the Pods:

```
$ kubectl version
$ kubectl get pods
```

Now, create your first Pod using the YAML file provided in the *Course Resources* (for a refresher on how to access the *Course Resources*, look at page 1.8): **`redis.yaml`**

```
$ kubectl create -f redis.yaml
```


Once the container is running, you will be able to access the logs of the container:

```
$ kubectl logs redis
```

And you will be able to *enter* the container and run the **redis** CLI:

```
$ kubectl exec -ti redis redis-cli
127.0.0.1:6379>
```

Create a Pod in a specific namespace

To avoid name collision, you can create *namespaces*. In each namespace, you can create Kubernetes resources. Let's create an **lfs248** namespace and create another **redis** pod in that namespace:

```
$ kubectl create ns lfs248
```

Check that your namespace has been created:

```
$ kubectl get ns
NAME          STATUS    AGE
default       Active    1d
kube-system   Active    1d
lfs248        Active    1m
```

You will see a **kube-system** namespace, which is created automatically by **minikube** and contains system specific services.

To create a Pod in a specific namespace, you need to specify it in the metadata of the Pod. For example, to create a **redis** Pod in the **lfs248** namespace. In the YAML file below, you see that the namespace in the metadata has changed from **default** to **lfs248**:

```
$ cat redis-lfs248.yaml
apiVersion: v1
kind: Pod
metadata:
  namespace: lfs248
  name: redis
spec:
  containers:
    - name: redis
      image: redis
```

Create the Pod and check that it is running in the correct namespace. You can then use the **logs** command by passing a **namespace** option:

```
$ kubectl create -f redis-lfs248.yaml
```



```
$ kubectl get ns
$ kubectl logs redis --namespace=lfs248
```

Explore kubectl

We just saw how to use **`kubectl`** to create Pods based on files, list pods and namespaces, access container logs and create namespaces. There is much more to **`kubectl`**, check the usage to see some of the commands available:

```
$ kubectl --help
```

Since all resources are managed via REST endpoints, you can use **`kubectl`** to delete the Pods and namespace you just created:

```
$ kubectl delete pods redis
$ kubectl delete pods redis --namespace=lfs248
$ kubectl delete ns lfs248
```

If you want to make the link with the API endpoints, check the very helpful **`kubectl --v=99`** verbose mode.

LAB 6

Objective: Create a deployment, scale it, trigger a rolling update and rollback. Understand how replica sets allow you to manage revisions.

Pods are the lowest compute unit of Kubernetes. To ensure that pods are always running, you define a **deployment**. A deployment is a declarative manifest that describes what Pods should be running and how many.

To create your first deployment, use the **`kubectl run`** command. It is a convenience wrapper to define single container Pods. Let's create a deployment for *Ghost,* the microblogging platform:

```
$ kubectl run ghost --image=ghost
$ kubectl get deployments
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
ghost     1         1         1            1           6s
```

With your deployment running, check that a corresponding Pod has been started:

```
$ kubectl get pods
NAME                      READY     STATUS    RESTARTS   AGE
ghost-943298627-kdhts     1/1       Running   0          9s
```

Now, scale your deployment. For example, let's scale it to five replicas:

```
$ kubectl scale deployments ghost --replicas=5
$ kubectl get pods
NAME                      READY     STATUS              RESTARTS   AGE
ghost-943298627-38761     0/1       ContainerCreating   0          2s
```



```
ghost-943298627-ghshb   0/1   ContainerCreating   0   2s
ghost-943298627-kdhts   1/1   Running             0   4m
ghost-943298627-slzdr   0/1   ContainerCreating   0   2s
ghost-943298627-xwljz   0/1   ContainerCreating   0   2s
```

To set the desired number of Pods, a replica sets resource has been automatically generated by the deployment. You can list this replica set with **`kubectl get rs`**. It continuously reconciles the desired state (e.g five replicas) with the actual cluster state. The replica set counts the number of Pods that matches the Pod selector defined in the deployment manifest and scales the Pods accordingly. Below, we show how you can parse the deployment manifest with **`jq`** to list the pod selection via labels.

```
$ kubectl get deployment ghost -o json | jq -r .spec.selector
{
  "matchLabels": {
    "run": "ghost"
  }
}
```

To see that the replica set does its job, delete one of the pods, and list Pods again. You will see that another one got started to still have the desired number of replicas running. You can also see this by removing the label from one of the running pods:

```
$ kubectl label pods ghost-943298627-38761 run-
$ kubectl get pods -Lrun
NAME                    READY   STATUS    RESTARTS   AGE   RUN
ghost-943298627-38761   1/1     Running   0          1h    <none>
ghost-943298627-ghshb   1/1     Running   0          1h    ghost
ghost-943298627-kdhts   1/1     Running   0          1h    ghost
ghost-943298627-nj122   1/1     Running   0          56m   ghost
ghost-943298627-slzdr   1/1     Running   0          1h    ghost
ghost-943298627-xwljz   1/1     Running   0          1h    ghost
```


Above you see that one Pod does not have the label **run=ghost**, because we removed it. The replica set automatically started a new Pod to have five running.

Explore a bit more the **kubectl labels** command and check how they can be set in resource metadata.

Rolling updates and rollbacks

One big advantage of deployments is that they can be used to do a rolling update of your Pods. For example, let's assume that you have a new image that you want to use. You can use the **kubectl set** command to change that image.

```
$ kubectl set image deployment/ghost ghost=ghost:0.9
```

List the replica sets again and see that a new set was created. This new set corresponds to the new image. The old and new replica sets will be scaled down and up respectively to keep the application running but provide the new image. At the end of the update, the new replica set has five running replicas and the old one has zero.

```
$ kubectl get rs
NAME               DESIRED   CURRENT   READY   AGE
ghost-3487275284   2         2         0       4s
ghost-943298627    4         4         4       1h
kubectl get rs
NAME               DESIRED   CURRENT   READY   AGE
ghost-3487275284   5         5         4       1m
ghost-943298627    0         0         0       1h
```

If you list your pods, you will see that the running pods now use the new replica set, except the pod that we re-labeled and was taken out of the replica set:

```
$ kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
ghost-3487275284-73lfn   1/1     Running   0          1m
ghost-3487275284-8pxvb   1/1     Running   0          2m
```



```
ghost-3487275284-pcbs5   1/1       Running   0          1m
ghost-3487275284-pnw4j   1/1       Running   0          2m
ghost-3487275284-t2b4q   1/1       Running   0          1m
ghost-943298627-38761    1/1       Running   0          1h
```

Now, check the history of your deployment:

```
$ kubectl rollout history deployment/ghost
deployments "ghost"
REVISION    CHANGE-CAUSE
1       <none>
2       <none>
```

You will see two revisions. The change-cause will be empty. You can record that cause by specifying the `--record` option when you create your deployment.

To rollback, simply do:

```
$ kubectl rollout undo deployment/ghost
```

If you keep an eye on your Pods, you will see that they will be terminated and new ones corresponding to the original replica set will be created. To learn more about deployments, do not forget to check the documentation.

LAB 7

Objective: Learn how to use volumes inside Pods. Learn how to create secrets and ConfigMaps and access them inside Pods.

This lab has three independent parts:

- Mount a volume in two containers via a Pod manifest
- Create a secret and use it to run a MySQL Pod
- Create a ConfigMap and mount it inside a Pod.

Using Volumes

Let's get started with volumes. You can do this part using the description below or skip to the step-by-step instructions.

- Write a single Pod manifest which contains two containers and one volume. Mount the volume in different paths in each of the containers and use **`kubectl exec`** to touch a file. Read the file from the other container. This will show you how to share data between containers in a Pod using a volume.

Let's create a Pod using the provided manifest **`volumes.yaml`** (retrieve it from *Course Resources*). It starts one Pod with two containers in it, one called **`busy`** and the other one called **`box`**.

```
$ kubectl create -f volumes.yaml
$ kubectl get pods
NAME       READY      STATUS     RESTARTS    AGE
vol        2/2        Running    0           22s
```

Connect inside the **busy** container, touch a file in the **busy** directory. Then, connect to the **box** container and see that same file in the **box** directory.

```
$ kubectl exec -ti vol -c busy -- touch /busy/lfs248
$ kubectl exec -ti vol -c box -- ls -l /box
total 0
-rw-r--r--    1 root     root             0 Dec 20 13:28 lfs248
```

A volume is being shared between the two containers in the Pod. The volume is mounted in different paths. Indeed, if you check the manifest **volumes.yaml** you see a *volumes* section common to the Pod, and *volumeMounts* sections specific to each container in the Pod. This is how volumes are accessed in containers.

```
...
spec:
  containers:
  - image: busybox
...
    volumeMounts:
    - mountPath: /busy
      name: test
    name: busy
  - image: busybox
...
    volumeMounts:
    - mountPath: /box
      name: test
    name: box
  volumes:
  - name: test
    emptyDir: {}
```


Using Secrets

- Launch a MySQL pod using a secret to store the MySQL root password. The MySQL image does not run without specifying this password. Create a secret with **`kubectl`** and then read this secret inside the Pod using an environment variable.

The Docker Hub official MySQL image needs to have a **`MYSQL_ROOT_PASSWORD`** environment variable set to run. In a Pod manifest, this looks like the snippet below:

```
spec:
  containers:
  - image: mysql:5.5
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: root
```

However, this is sub-optimal, as it shows the value of the password in the actual manifest. It would be better to create a secret and bind that secret to the Pod at runtime. We can do this. First create a secret by hand with **`kubectl`**:

```
$ kubectl create secret generic mysql --from-literal=password=root
```

You can then use this secret inside a Pod like so:

```
spec:
  containers:
  - image: mysql:5.5
    env:
    - name: MYSQL_ROOT_PASSWORD
      valueFrom:
        secretKeyRef:
          name: mysql
          key: password
```

Use the manifest provided (in the *Course Resources*) to create the MySQL Pod and enter the container to check that the database is running:

```
$ kubectl exec -ti mysql -- mysql -uroot -proot
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4
Server version: 5.5.54 MySQL Community Server (GPL)

Copyright (c) 2000, 2016, Oracle and/or its affiliates. All rights
reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.

mysql>
```

Using ConfigMaps

- Create a ConfigMap with `kubectl` and mount it inside a Pod. Then, get inside the container and verify that the file is there.

```
$ kubectl create configmap map --from-file=configmap.md
configmap "map" created
$ kubectl get configmaps
NAME       DATA       AGE
map        1          5s
```


Use the **configmap.yaml** file (retrieve it from *Course Resources*) to create the Pod. If you check the content of the **configmap.md** file (retrieve it from *Course Resources*) inside the container, you will see that it corresponds to the original file:

```
$ kubectl create -f configmap.yaml
$ kubectl exec -ti configmap-test -- ls /config
configmap.md
```

Using the ConfigMaps resource, we can share files between containers and load configurations inside containers.

A ConfigMap can be mounted as a volume inside a Pod, just like a regular volume. There are additional ways to refer to a ConfigMap, especially using environment variables.

```
...
spec:
  containers:
  - image: busybox
...
    volumeMounts:
    - mountPath: /config
      name: map
    name: busy
  volumes:
    - name: map
      configMap:
        name: map
```


LAB 8

Objective: Create a single **nginx** Pod, create a service that selects this Pod, and access it from your browser. Verify that the endpoint has been populated and that DNS resolves the service name.

Services define a virtual IP address that directs traffic to Pods that match a label selector.

Let's create a Pod that runs **nginx**. The manifest is in the **nginx.yaml** file (retrieve it from *Course Resources*).

```
cat nginx.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: default
  labels:
    app: nginx
spec:
  containers:
  - image: nginx
    ports:
      - containerPort: 80
    imagePullPolicy: IfNotPresent
    name: nginx
```

Create it with **kubectl**


```
$ kubectl create -f nginx.yaml
```

The service is defined in another manifest, **`nginx-svc.yaml`** (retrieve it from *Course Resources*). The selector uses the label defined in the Pod manifest **`app: nginx`**. The port definition sets port 80 as the container port to reach the application.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  ports:
    - port: 80
  type: NodePort
  selector:
    app: nginx
```

Once you create the service, you will see the IP of the matching Pod in the endpoints list:

```
$ kubectl create -f nginx-svc.yaml
$ kubectl get svc
NAME         CLUSTER-IP    EXTERNAL-IP    PORT(S)    AGE
nginx        10.0.0.162    <nodes>        80/TCP     3s
$ kubectl get endpoints
NAME         ENDPOINTS         AGE
nginx        172.17.0.4:80     6s
```

Reach the service with your browser by typing **`minikube service nginx`**.

Finally, verify that DNS is working. Creating a *sleeping* **`busybox`** container, and *exec* into it to run **`nslookup`**:

```
$ kubectl create -f busybox.yaml
$ kubectl exec -ti busybox -- nslookup nginx
Server:    10.0.0.10
```



```
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      nginx
Address 1: 10.0.0.162 nginx.default.svc.cluster.local
```

In the example above, the IP 10.0.0.162 returned by the DNS lookup corresponds to the service IP address.

LAB 9

Objective: Deploy an Ingress controller, create an Ingress rule to reach a clusterIP type service.

The manifest for an Ingress controller that works on **`minikube`** is in your lab folder (retrieve from *Course Resources*) and is called **`backend.yaml`**. Just create the controller with **`kubectl`**:

```
$ kubectl create -f backend.yaml
```

With the Ingress controller now running, you will repeat part of the lab from Chapter 8 and create an Ingress rule as well.

To make this straightforward, check the manifest **`nginx.yaml`**. You will see the manifest for a Pod, a service that matches the pod labels, and an Ingress rule. Create it.

```
$ kubectl create -f nginx.yaml
```

You will now see a running pod, a service, and an Ingress rule.

```
$ kubectl get ingress
NAME     HOSTS                          ADDRESS          PORTS     AGE
nginx    nginx.192.168.99.100.nip.io    192.168.99.100   80        3m
```

The manifest describing the Ingress is shown below:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: nginx
spec:
```



```yaml
rules:
- host: nginx.192.168.99.100.nip.io
  http:
    paths:
    - backend:
        serviceName: nginx
        servicePort: 80
```

This Ingress rule is used by the Ingress controller (started by the **`backend.yaml`** manifest) to re-configure the nginx proxy running on the head node (in our case, **`minikube`**). The rule will proxy requests for host **`nginx.192.168.99.100.nip.io`** to the internal service called **`nginx`**.

We use the nip.io service. It is a wildcard DNS service that is very handy for testing. It will resolve **`nginx.192.168.99.100.nip.io`** to **`192.168.99.100`** the IP of minikube. Note that you may need to edit the Ingress rule manifest if the IP of your minikube is different.

Once the rule is implemented by the controller (could take O(10) s), open your browser at **`nginx.192.168.99.100.nip.io`** and enjoy **`nginx`**.

Try to reproduce this with a **`ghost`** application.

LAB 10

Objective: Launch a DaemonSet, learn a few additional Pod specification parameters.

A DaemonSet is a **`v1beta1`** extension API resource. A DaemonSet will start one Pod on each node in your cluster (or a set of Pods selected by labels).

DaemonSets are used to run services daemons such as monitoring or network daemons. For example, in conjunction with **`kubeadm`**, you can use them to create a network overlay when building your cluster.

In your lab folder (in the *Course Resources*) you will find a basic DaemonSet manifest **`busy-daemon.yaml`**, it will run a sleeping **`busybox`** Pod with some special pod specification:

```
$ cat busy-daemon.yaml
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: busy-daemon
spec:
  template:
    metadata:
      labels:
        name: busy-daemon
    spec:
      hostNetwork: true
      hostPID: true
      containers:
```



```
    - name: busybox
      image: busybox
      command:
        - sleep
        - "3600"
      securityContext:
        privileged: true
```

Once you create it, you will see a Pod running in your minikube. Since minikube only has one node, you will not see other Pods. That Pod is special, as it has the network namespace of the host, the process namespace of the host and is a privileged container. This is a fairly *dangerous* container.

```
$ kubectl get daemonset
NAME          DESIRED   CURRENT   NODE-SELECTOR   AGE
busy-daemon   1         1         <none>          6s
$ kubectl get pods
NAME                               READY     STATUS              RESTARTS
AGE
busy-daemon-m3bcc                  0/1       ContainerCreating   0
2s
```

Verify that it has the network configuration of the host:

```
$ kubectl exec -ti busy-daemon-m3bcc -- ifconfig
```

Verify that it has the process namespace of the host:

```
$ kubectl exec -ti busy-daemon-m3bcc -- ps -ef | grep docker
```

With such a Pod you can imagine running all your cluster daemons the same way that you now run distributed applications on Kubernetes.

LAB 11

Objective: Explore and practice scheduling features.

This lab consists of three small experiments to highlight basic scheduling features.

Create a Pod using the **`redis.yaml`** manifest provided.The Pod will get scheduled automatically, the default scheduler does its job, and your Pod starts after the image gets downloaded.

Create a Pod using the **`redis-selector.yaml`** manifest provided. Note the **`nodeSelector`** field, which targets a node with the label **`foo=bar`**. You should see that the Pod remains in *pending* state. There are no nodes that match the **`nodeSelector`** labels.

Confirm this with:

```
$ kubectl describe pods foobar-node
```

How can you get this Pod scheduled?

Hint:

```
$ kubectl label node minikube foo=bar
```

Create a Pod using the **`redis-sched.yaml`** manifest provided. Note the **`schedulerName`** field in the manifest.

What happens? Why?

To get this Pod scheduled, you can run another scheduler. You can do this by starting the same container image like the default scheduler by setting the scheduler name option to **`foobar`**. For more details, please see this guide. You can also create a Pod Binding by hand.

Check the **`binding.json`** file in your lab directory. It basically defines a Node target **minikube** for the pod **`foobar-sched`**. By attaching this Binding to the Pod object, we will schedule the Pod by hand on **minikube**.

For simplicity, run a proxy in a separate terminal **`kubectl proxy --port=8080`**, then use **`curl`** to **`POST`** the Binding:

```
$ curl -H "Content-Type:application/json" -X POST --data @binding.json
http://localhost:8080/api/v1/namespaces/default/pods/foobar-sched/binding
/
```

Note the API endpoint of the Binding, referring to your Pod name:
`api/v1/namespaces/default/pods/foobar-sched/binding/`.

LAB 13

Objective: Create your first Third-Party API resource and create a corresponding custom resource.

A third-party API resource is a powerful API resource that will allow you to let Kubernetes create REST endpoints for new resources that you will create. In this lab we will create one, but we will not create a *controller* that will be able to watch these resources and act on them.

```
apiVersion: extensions/v1beta1
kind: ThirdPartyResource
metadata:
  name: pin-guin.k8s.lfs258.com
description: "A crazy pinguin for LSF258"
versions:
- name: v1
```

Create it and list it:

```
$ kubectl create -f pinguins.yml
$ kubectl get thirdpartyresource
NAME                         DESCRIPTION                      VERSION(S)
pin-guin.k8s.lfs258.com      A crazy pinguin for Kubernetes   v1
```

With this setup, a new API group has been created. You can now create a custom resource of type `PinGuin`. Check the following manifest in your folder (*Course Resources*):

```
apiVersion: k8s.lfs258.com/v1
kind: PinGuin
metadata:
```



```
  name: crazy
  labels:
    kubernetes: rocks
```

Create it and verify that **kubectl** has discovered it.

```
$ kubectl create -f pinguin.yml
$ kubectl get pinguins
$ kubectl get pinguins
NAME      LABELS              DATA
crazy     kubernetes=rocks
{"apiVersion":"k8s.lfs258.com/v1","kind":"PinGuin"...
```

You now have a new REST endpoint for **pinguins**. You can watch these in a custom built controller and perform actions based on *pinguins* events.

LAB 15

Objective: Install Helm and launch an application packaged as a Chart.

Download Helm from the GitHub release page. Verify that the **helm** binary is in your PATH:

```
$ helm version
```

From your local machine, you can setup Helm on minikube. **helm init** will launch a **tiller-deploy** deployment.

```
$ helm init
$ kubectl get deployment --all-namespaces
NAMESPACE     NAME            DESIRED   CURRENT   UP-TO-DATE   AVAILABLE
AGE
kube-system   tiller-deploy   1         1         1            1
2m
```

You are ready to search for Charts and install them in your cluster. Let's search for *redis* and install the corresponding chart. We will not use persistence for this exercise.

```
$ helm search redis
NAME                          VERSION DESCRIPTION
stable/redis                  0.4.2   Chart for Redis
$ helm install stable/redis --set persistence.enabled=false
$ helm list
NAME            REVISION    UPDATED                     STATUS      CHART
```

```
misty-lionfish  1          Tue Dec 20 19:01:18 2016    DEPLOYED
redis-0.4.2
```

With a deployed Chart, you should see a *redis* pod being created.

```
$ kubectl get pods
NAME                                  READY     STATUS    RESTARTS    AGE
misty-lionfish-redis-3119530502-3q0xl   1/1       Running   0           10s
```

Helm uses ConfigMaps to store each release information. Check your ConfigMaps. You can then delete your release.

```
$ kubectl get configmap --all-namespaces
NAMESPACE     NAME                     DATA      AGE
kube-system   misty-lionfish.v1        1         5m
$ helm delete misty-lionfish
```

This is just the quickest exercise with Helm. You can now dive into writing your own Chart, setting up your own repository and using Helm to do rolling updates of entire applications.

