# Table of Contents

Lab Overview - HOL-1831-01-CNA - Kubernetes - Getting Started ........................................ 2  
Lab Guidance .................................................................................................................. 3  
Module 1 - Introduction to Kubernetes (30 minutes) ......................................................... 9  
   Introduction .................................................................................................................. 10  
   What is container orchestration and why do I need it? ............................................. 12  
   Terminology is a barrier. Kubernetes objects explained ........................................... 14  
   Kubernetes Architecture Deep Dive ........................................................................... 23  
   Conclusion .................................................................................................................. 26  
Module 2 - Kubernetes Deep Dive (45 minutes) ............................................................... 28  
   Your Lab Kubernetes Cluster .................................................................................... 29  
   Namespaces and CLI context ................................................................................... 35  
   Deployments, Pods and Services ............................................................................. 38  
   Private Registry With Harbor ................................................................................... 49  
   Conclusion .................................................................................................................. 57  
Module 3 - Deploy and Manage a Multi-Tiered Application (30 minutes) ....................... 59  
   Introduction ................................................................................................................ 60  
   Deploy and Upgrade Hobby Shop Review Application ........................................... 61  
   Conclusion .................................................................................................................. 84
Lab Overview - HOL-1831-01-CNA - Kubernetes - Getting Started
Lab Guidance

Note: It will take more than 90 minutes to complete this lab. The modules are independent of each other so you can start at the beginning of either module and proceed from there. You can use the Table of Contents to access the module of your choosing.

The Table of Contents can be accessed in the upper right-hand corner of the Lab Manual.

[Lab Abstract: Kubernetes is fast becoming the standard for enterprise container orchestration. In this lab you will be exposed to the fundamentals of the Kubernetes architecture and deep dive into using the kubectl CLI. You will also dive into the details of the building, deployment and management of container based applications on Kubernetes. Kubernetes is an open-source platform for automating deployment, scaling, and operations of application containers across clusters of hosts, providing container-centric infrastructure.

With Kubernetes, you are able to quickly and efficiently respond to customer demand:

• Deploy your applications quickly and predictably.
• Scale your applications on the fly.
• Seamlessly roll out new features

Lab Module List:

• **Module 1 - Introduction to Kubernetes** (30 minutes) (Basic) In Module 1 you will learn what container orchestration with Kubernetes is all about. Terminology is a barrier to understanding technology and the fundamental components of the Kubernetes platform will be explained in detail. This module contains no lab steps but will provide you with a foundational understanding that will make the other modules more intuitive.

• **Module 2 - Kubernetes Deep Dive** (45 minutes) (Advanced) This module is pure Kubernetes lab work. You will use the kubectl CLI to manipulate the Kubernetes components you read about in Module 1. You will create the deployment that defines the pods and services that make up your nginx webservice application. You will scale the application both up and down - adding pods to provide either increased capacity or availability. By the end you should be very comfortable with manipulating your application components through the CLI.

• **Module 3 - Deploy and Manage a Multi-tiered Application** (30 minutes) (Advanced) This module can be taken independently from the first two, but builds on the knowledge gained there to deploy a more complex application. You will see the components of a web-based Hobby shop product rating system application that includes a backend database. Once your application is deployed,
you will upgrade to a newer version with no downtime, and will subsequently roll the upgrade back - again without downtime.

Lab Captain:

- Michael West, Technical Product Manager, Cloud Native BU, USA

This lab manual can be downloaded from the Hands-on Labs Document site found here:

http://docs.hol.vmware.com

This lab may be available in other languages. To set your language preference and have a localized manual deployed with your lab, you may utilize this document to help guide you through the process:


**Location of the Main Console**

1. The area in the RED box contains the Main Console. The Lab Manual is on the tab to the Right of the Main Console.
2. A particular lab may have additional consoles found on separate tabs in the upper left. You will be directed to open another specific console if needed.
3. Your lab starts with 90 minutes on the timer. The lab can not be saved. All your work must be done during the lab session. But you can click the **EXTEND** to
increase your time. If you are at a VMware event, you can extend your lab time twice, for up to 30 minutes. Each click gives you an additional 15 minutes. Outside of VMware events, you can extend your lab time up to 9 hours and 30 minutes. Each click gives you an additional hour.

Alternate Methods of Keyboard Data Entry

During this module, you will input text into the Main Console. Besides directly typing it in, there are two very helpful methods of entering data which make it easier to enter complex data.

Click and Drag Lab Manual Content Into Console Active Window

You can also click and drag text and Command Line Interface (CLI) commands directly from the Lab Manual into the active window in the Main Console.

Accessing the Online International Keyboard

You can also use the Online International Keyboard found in the Main Console.

1. Click on the Keyboard Icon found on the Windows Quick Launch Task Bar.
Click once in active console window

In this example, you will use the Online Keyboard to enter the "@" sign used in email addresses. The "@" sign is Shift-2 on US keyboard layouts.

1. Click once in the active console window.
2. Click on the **Shift** key.

Click on the @ key

1. Click on the **"@ key"**.

Notice the @ sign entered in the active console window.
Activation Prompt or Watermark

When you first start your lab, you may notice a watermark on the desktop indicating that Windows is not activated.

One of the major benefits of virtualization is that virtual machines can be moved and run on any platform. The Hands-on Labs utilizes this benefit and we are able to run the labs out of multiple datacenters. However, these datacenters may not have identical processors, which triggers a Microsoft activation check through the Internet.

Rest assured, VMware and the Hands-on Labs are in full compliance with Microsoft licensing requirements. The lab that you are using is a self-contained pod and does not have full access to the Internet, which is required for Windows to verify the activation. Without full access to the Internet, this automated process fails and you see this watermark.

This cosmetic issue has no effect on your lab.

Look at the lower right portion of the screen
Please check to see that your lab has finished all the startup routines and is ready for you to start. If you see anything other than "Ready", please wait a few minutes. If after 5 minutes your lab has not changed to "Ready", please ask for assistance.
Module 1 - Introduction to Kubernetes (30 minutes)
Introduction

What is Kubernetes?

Kubernetes is rapidly becoming the leading platform for managing cloud native, microservice based applications. Container orchestration is a critical element in modern applications developed using DevOps practices. Kubernetes provides all of the constructs out of the box for a service that:

- Comprises a variable number of pods (units of compute) each of which consumes a fixed amount of CPU and memory
- Consumes compute resources elastically as it scales up and down
- Stores data on a networked persistent disk
- Discovers other services via DNS
- Is exposed externally via a load balancer.

Now with existing vSphere infrastructure users can directly support infrastructure consumption via Kubernetes and provide an enterprise-tested platform for modern cloud-native workloads.

This Module contains the following lessons:

- [Lesson 1] What is container orchestration and why do I need it?
- [Lesson 2] Terminology is a barrier. Kubernetes objects explained
- [Lesson 3] Kubernetes Architecture Deep Dive
Note: Module 1 is all reading and goes into some depth on the terminology and architecture. If at any time, you feel this is more than you need, please jump to Module 2 for hands on with Kubernetes.
What is container orchestration and why do I need it?

Your organization drank the Docker Kool-Aid. Developers started building containers on their desktops. They found that curated base images available on Dockerhub were a fantastic way to jumpstart application development. They started to create development pipelines that were container based. Deployment was a problem because it was still largely a manual process, so they started breaking applications up in to smaller and smaller components. You might call this a micro-service or not - but the implementation is through containers. Suddenly, your organization is running 100’s of containers - or more.

Developers aren't quite sure how to operationalize all of these disparate container workloads, but they do know that automated orchestration is the key.

What does that mean?

**Container Scheduling**: Containers need to be distributed across container hosts in a way that levels the use of host resources. Virtual Machine placement on vSphere hosts can be handled by the Distributed Resource Scheduler (DRS). A similar capability is needed for containers. The physical resources need isolation capability - the ability to define availability zones or regions. Affinity and anti-affinity become important. Some workloads must run in close proximity to others - or to provide availability, must run on separate physical hosts.

**Container Management**: The ecosystem of tools available to the operations team today tend to stop at the host operating system - without providing views into the containers themselves. These tools are becoming available, but are not yet widely adopted. Monitoring of running container applications and recovery upon failure must be addressed. Container images need to be managed. Teams need a mechanism for image isolation, such as role based access control and signing of content. Image upgrade and rollout to running applications must be addressed. Orchestration must also include the capability to scale the application up or down to provide for changes in resource consumption or availability requirements.

**Service Endpoints**: Containers are ephemeral. They are short lived and are expected to die. When they restart or are recreated, how do other applications find them? Service Discovery is critical to operationalizing containers at scale. Service Endpoints need to be redundant and support Load Balancing. They should also auto scale as workloads increase.

**External Endpoints**: Not all container based applications are entirely deployed in containers and many must persist application state. There is a need to access external resources like databases - or to configure and manage software defined networking.
Persistent volume support is needed for those applications that need to retain state even when the container-based components fail.

Kubernetes is an open-source platform for automating deployment, scaling, and operations of application containers across clusters of hosts, providing container-centric infrastructure.

**Capabilities:**

- Deploy your applications quickly and predictably
- Scale your applications on the fly
- Seamlessly roll out new features
- Optimize use of your hardware by using only the resources you need
Terminology is a barrier. Kubernetes objects explained

Many people new to the container space and Kubernetes get hung up on all of the new terminology. Before jumping into the details of the platform, we are going to spend a little time defining some of the terms that will be used later on to describe the function of the platform. The goal is to provide some level of depth on these topics, however if you find that this is more than you need, skip to Module 2 and start using Kubernetes.

**Kubernetes Cluster**

A cluster is very simply the physical or virtual machines and other infrastructure resources used by Kubernetes to run your applications. You define a set of machines, create networking and attach storage, then install the Kubernetes system services. Now you have a running cluster. This does not mean that there is any sort of traditional clustering technology in the infrastructure sense - nor does it align with vSphere clustering constructs. That has been a point of confusion for many VMware administrators. A cluster is simply a set of VMs, wired together, with attached local or shared storage - and running the Kubernetes System services.

**Kubernetes Node**

A node is any of the physical machines or VMs that make up the Kubernetes cluster. Nodes are of two types: Master (sometimes called Leader) and Worker. Some Master
based services can be broken out into their own set of VMs and would also be referred to as nodes (we will get to Etcd shortly). Master nodes run the kube-system services. The Worker nodes run an agent and networking proxy, but are primarily thought of as the set of nodes that run the pods.

**Pods**

![Diagram of a pod showing Nginx, mgmt, and logging containers]

Pods are the smallest deployable units of computing that can be created and managed in Kubernetes. Pods are always co-located and co-scheduled, and run in a shared context. A pod models an application-specific logical host - it contains one or more application containers which are relatively tightly coupled. The shared context of a pod is a set of Linux namespaces, cgroups, and potentially other facets of isolation - the same things that isolate a Docker container.

In this sample pod, there are three application containers. The Nginx webserver, along with ssh and logging daemons. In a non-container deployment, all three of these would probably run as individual processes on a single VM. Containers generally run a single process to keep them lightweight and avoid the need for init configuration. Notice in the image that there is also a Pause container. This container actually hosts the networking stack, the other three containers will share the IP and listen on different ports. This allows all containers in a pod to communicate via localhost. Notice that the pod in this example has a single IP: 10.24.0.2 on a network that is generally private to the Kubernetes cluster. The pod is a logical abstraction that is managed by Kubernetes. If you log onto a Kubernetes node VM and look for pods, you won't find them through Docker. You will be able to see a set of containers, but no pods. You will find the pods through the Kubernetes CLI or UI.
Replica Sets

A Replica Set ensures that a specified number of pod replicas are running at any given time. A replication controller process watches the current state of pods and matches that with the desired state specified in the pod declaration. If there is a difference, because a pod has exited, it attempts to make the desired state and current state consistent by starting another pod. Developers may choose to define replica sets to provide application availability and/or scalability. This definition is handled through a configuration file defined in yaml or json syntax.

Services

Kubernetes pods are ephemeral. They are created and when they die, they are recreated - not restarted. While each pod gets its own IP address, even those IP addresses cannot be relied upon to be stable over time. This leads to a problem: if some set of pods - like Redis slave (Redis is a Key/Value store with Master/Slave architecture) - provides functionality to other pods - like a frontend Webserver - inside the Kubernetes cluster, how do those frontends find and keep track of which backends are in that set?

Enter Services.

A Kubernetes Service is an abstraction which defines a logical set of pods and a policy by which to access them - sometimes called a micro-service. The set of pods targeted
by a service is (usually) determined by a label selector (Explained on the next page). A
service generally defines a ClusterIP and port for access and provides East/West Load
Balancing across the underlying pods.

Let's look at this in the context of the diagram above. There are two Redis-slave pods -
each with its own IP (10.24.0.5, 10.24.2.7). When the service is created, it is told that
all pods with the label Redis-slave are part of the service. The IPs are updated in the
endpoints object for the service. Now when another object references the service
(through either the service clusterIP (172.30.0.24) or its DNS entry, it can load balance
the request across the set of pods. Kubernetes includes its own DNS for internal domain
lookups and each service has a record based on its name (redis-slave).

To this point we have only talked about internal access to the service. What if the
service is a web server and users must access it from outside the cluster. Remember
that the IPs aren't routable outside the private cluster overlay network. In that case
there are several options - Ingress Servers, North/South Load Balancing, and NodePort.
We will discuss NodePort here because that is what will be used in the lab. In the service
declaration, a specification of type NodePort means that each cluster node will be
configured so that a single port is exposed for this service. So a user could get access
to the frontend web service in the diagram by specifying the IP address of any node in
the cluster, along with the NodePort for the frontend service. The service then provides
East/West load balancing across the pods that make up the service.

Labels and Selectors

The esoteric definition is as follows:

- Key:Value pairs that can be attached to any Kubernetes object (pods, nodes, services)
- Ex: Identify releases (Beta, Prod), Environments (Dev, Prod), Tiers (Frontend, Backend)
- Selectors are the mechanism for group filtering based on the labels

A more straightforward way to say this is Kubernetes is architected to take action on
sets of objects. The sets of objects that a particular action might occur on are defined
through labels. We just saw one example of that where a service knows the set of pods
associated with it because a selector (like run:redis-slave) was defined on it and a set of
pods was defined with a label of run:redis-slave. This methodology is used throughout
Kubernetes to group objects.

Deployments

A deployment is a declarative object for defining your desired Kubernetes application
state. It includes the number of replicas and handles the roll-out of application updates.
Deployments provide declarative updates for pods and replica sets (the next-generation
replication controller). You only need to describe the desired state in a deployment
object, and the deployment controller will change the actual state to the desired state at
a controlled rate for you. Think of it as a single object that can, among other things, define a set of pods and the number of replicas, while supporting upgrade/rollback of pod image versions.

Namespaces

Namespaces are intended for use in environments with many users spread across multiple teams, or projects. Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces. They are a way to divide cluster resources between multiple uses. As Kubernetes continues to evolve, namespaces will provide true multi-tenancy for your cluster. They are only partially there at this point. By default, all resources in a Kubernetes cluster are created in a default namespace. A pod will run with unbounded CPU and memory requests/limits. A Kubernetes Namespace allows users to partition created resources into a logically named group. Each namespace provides:

- a unique scope for resources to avoid name collisions
- policies to ensure appropriate authority to trusted users
- ability to specify constraints for resource consumption

This allows a Kubernetes cluster to share resources by multiple groups and provide different levels of QoS to each group. Resources created in one namespace are hidden from other namespaces. Multiple namespaces can be created, each potentially with different constraints. You will see how namespaces are used in Module 2
Load balancing in Kubernetes can be a bit of a confusing topic. The Kubernetes cluster section shows an image with load balancers. Those represent balancing requests to the Kubernetes control plane. Specifically the API Server. But what if you deploy a set of pods and need to load balance access to them? We have previously discussed services. In addition to discovery, services also provide load balancing of requests across the set of pods that make up the service. This is known as East/West load balancing and is internal to the cluster. If there is a need for ingress to a service from an external network, and a requirement to load balance that access, this is known as North/South load balancing. There are three primary implementation options:

- Create service with type ‘LoadBalancer’. This is platform dependent and requires that the load balancer distributing inbound traffic is created through an external load balancer service (Like AWS ElasticLB). Generally this incurs a cost from the cloud platform and may be somewhat inflexible.
- Statically configure an external load balancer (Like F5) that sends traffic to a K8s Service over ‘NodePort’ on specific nodes. In this case, the configuration is done directly on the external load balancer after the service is created and the nodeport is known.
- Create Kubernetes Ingress ; This is a Kubernetes object that describes a North/South load balancer. The Kubernetes ingress object is ‘watched’ by an ingress
controller that configures the load balancer datapath. Usually both the ingress controller and the load balancer datapath are running as pods. This requires that an ingress controller be created, but may be the most flexible solution. NSX-T provides an ingress controller.

## Sample Application

This application contains three separate deployments—Frontend, Redis Master and Redis Worker. A deployment provides a declarative method for defining pods, replica sets and other Kubernetes constructs. The Frontend Deployment includes a Frontend pod, which runs an Nginx Webserver. It defines a replica set that maintains three running copies of the Frontend pod. It also defines a Frontend service that provides an abstraction to the underlying Frontend pods, including a ClusterIP and nodeport that can be used to access the service. The Frontend deployment also defines a persistent storage volume that allows stateful application data to be stored and persisted across pod failures.

The application is also using a Redis Key:Value store for its data. Redis' architecture is single Master with multiple Worker nodes. The Master and Workers are separated into their own deployments, with their own replica sets and services. Let's now dig into the configuration files that would be needed to define this application.

## Yaml Files

The files for creating the deployments and their services can be in yaml or json format. Usually yaml is used because it is easier to read. Below are the yaml files used to create the frontend deployment and the frontend service. The other yaml files are available as part of module 3. Note that though persistent volumes are shown in the
diagram, they were not used in this lab and are not part of the configuration in the yaml files.

**Frontend Deployment**

```yaml
apiVersion: extensions/v1beta1
class: Deployment
metadata:
  name: frontend
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: guestbook
tier: frontend
    spec:
      containers:
        - name: php-redis
          image: gb-frontend:v5
          resources:
            requests:
              cpu: 10m
              memory: 100Mi
          env:
            - name: GET_HOSTS_FROM
              value: dns
          ports:
            - containerPort: 80
```

- **What kind of object is being defined**
- **Give it a name**
- **Run three copies of the pod**
- **Labels and Selectors**
- **Run the gb-frontend:v5 container**
- **Pass Environment Variables to Container**
- **Listen on Container port 80**
This file defines the deployment specification. Think of it as the desired state for the deployment. It has a name - frontend. It defines a replica set that includes 3 replicas. That means the desired state for this deployment is that 3 copies of every pod is running. Labels are defined for these pods. You will see below that the service definition will use these to define the pods that are covered by the service. The container in the pod will be based on the gb-frontend:v5 image. The lab images are stored in a local Harbor registry so you will notice a different image path in the lab. Resources can be constrained for the container based on the requests Key. Env: defines a set of environment variables that are passed to the container. Lastly, the container will be listening on port 80. Remember that this is container port 80 and must be mapped to some host port in order to access it from an external network.

This file defines the frontend service specification. The important pieces are the Type: Nodeport and the Selector. Specifying Type: Nodeport means that each Kubernetes cluster node will expose the same port (in the 30000 range) for access to this service. The service will then route requests to one of the pods that has a label from the service's selector. So all pods with labels app:guestbook or tier:frontend will be included in this service.
Kubernetes Architecture Deep Dive

At a very high level, the Kubernetes cluster contains a set of Master services that may be contained in a single VM or broken out into multiple VMs. The Master includes the Kubernetes API, which is a set of services used for all internal and external communications. Etcd is a distributed key value store that holds all persistent meta data for the Kubernetes cluster. The scheduler is a Master service that is responsible for scheduling container workloads onto the Worker nodes. Worker nodes are VMs that are placed across ESXi hosts. Your applications run as a set of containers on the worker nodes. Kubernetes defines a container abstraction called a pod, which can include one or more containers. Worker nodes run the Kubernetes agent, called Kubelet, which proxies calls to the container runtime daemon (Docker or others) for container create/stop/start/etc. etcd provides an interesting capability for "Watches" to be defined on its data so that any service that must act when meta data changes simply watches that key:value and takes its appropriate action.

A Kubernetes cluster can have one or more master VMs and generally will have etcd deployed redundantly across three VMs.

- **API Server**: Target for all operations to the data model. External API clients like the Kubernetes CLI client, the dashboard Web-Service, as well as all external and internal components interact with the API Server by ‘watching’ and ‘setting’ resources.
• **Scheduler**: Monitors container (pod) resources on the API Server, and assigns Worker nodes to run the pods based on filters

• **Controller Manager**: Embeds the core control loops shipped with Kubernetes. In Kubernetes, a controller is a control loop that watches the shared state of the cluster through the API Server and makes changes attempting to move the current state towards the desired state

• **Etcd**: Is used as the distributed key-value store of Kubernetes

• **Watching**: In etcd and Kubernetes everything is centered around ‘watching’ resources. Every resource can be watched on etcd through the API Server

• **Kubelet**: The Kubelet agent on the nodes is watching for ‘PodSpecs’ to determine what it is supposed to run and Instructs container runtimes to run containers through the container runtime API interface. PodSpecs are defined through the yaml configuration files seen earlier.

• **Docker**: Is the most used container runtime in Kubernetes. However K8s is ‘runtime agnostic’, and the goal is to support any runtime through a standard interface (CRI-O)

• **Rkt**: Besides Docker, Rkt by CoreOS is the most visible alternative, and CoreOS drives a lot of standards like CNI and CRI-O (Check out [https://www.cncf.io/](https://www.cncf.io/) for more on these standards)

• **Kube-Proxy**: Is a daemon watching the K8s ‘services’ on the API Server and implements east/west load-balancing on the nodes using NAT in IPTables
Let's look at a sample workflow. This is a high level view and may not represent the exact workflow, but is a close approximation. A user wants to create a pod through the CLI, UI or using the API through their own code. The request comes to the Kubernetes API Server. The API Server instantiates a pod object and updates etcd with the information. The scheduler is watching for pod objects that have no node associated with it. The scheduler sees the new pod object and goes through its algorithm for finding a node to place the pod (available resources, node selector criteria, etc.). Scheduler updates the pod information (through the API Server) to include the placement node. On that node, Kubelet is watching etcd for a pod object that contains its node. Once it sees the new pod object, it begins to instantiate the pod. Kubelet will call the container runtime engine to instantiate the set of containers that make up the pod. Once the pod is running and has an IP address, that information is updated in etcd so that the new Endpoint can be found.

Now that you know a little about how Kubernetes works, move on to Module 2 and try it out.
Conclusion

You should now have an understanding of the Kubernetes architecture and the fundamental terminology of the product. Now let's use it!

You've finished Module 1

Congratulations on completing Module 1.

Proceed to any module below which interests you most.

- **Module 2 - Kubernetes Deep Dive** (45 minutes) (Advanced) This module is pure Kubernetes lab work. You will use the kubectl CLI to manipulate the Kubernetes components you read about in Module 1. You will create the deployment that defines the pods and services that make up your nginx webserver application. You will scale the application both up and down - adding pods to provide either increased capacity or availability. By the end you should be very comfortable with manipulating your application components through the CLI.
- **Module 3 - Deploy and Manage Multi-tiered Application** (30 minutes) (Advanced) This module can be taken independently from the first two, but builds on the knowledge gained there to deploy a more complex application. You will see the components of a web-based Hobby shop product rating system application that includes a backend database. Once your application is deployed, you will upgrade
to a newer version with no downtime, and will subsequently roll the upgrade back - again without downtime.

**How to End Lab**

![End Lab Button](image)

To end your lab click on the **END** button.
Module 2 - Kubernetes Deep Dive (45 minutes)
Your Lab Kubernetes Cluster

The command line tool used to interact with Kubernetes clusters is kubectl. While you can use curl and other programs to communicate with Kubernetes at the API level, the kubectl command makes interacting with the cluster from the command line easy, packaging up your requests and making the API calls for you. In this section you will become familiar with some of the basic kubectl commands and get comfortable with a few of the constructs we described in the overview section. You will focus on system level components before moving on to applications. The lab contains a previously deployed Kubernetes cluster. The cluster contains three nodes - one master and two workers. Let's take a look at what we have deployed

Connect to vCenter

1) Click on Google Chrome

2) Click on vcsa-01a Bookmark and choose your vcenter web client

3) Choose your vcenter web client
The Web Client (Flash) version was chosen for the lab manual, but you might want to try the html client.
Verify all VMs are Running

You will notice that there are 4 VMs in the RegionA01-COMP01 cluster. The Master and Worker nodes for your Kubernetes cluster, as well as the Harbor VM. Harbor is VMware's container registry and is storing all of the container images used in this lab. More on that later.

1) Please verify that all 4 of these VMs are running.
Connect to Kubernetes Cluster

You are now going to ssh into the Kubernetes Master VM using Putty. For the purpose of this lab we are running the Kubernetes CLI (kubectl) in the cluster Master VM. We could just as easily run it on any other client.

1) Click on Putty from your Windows Desktop

2) Select K8 Master

3) Click Open
Check Cluster Components

Let's start getting familiar with using the Kubernetes CLI. You will start using the "get" command to view system level components of your Kubernetes cluster.

1) `kubectl get nodes`

View the availability of each of the nodes in your cluster and verify that each node is in "Ready" status.

2) `kubectl get cs`

View the status of the system components. The scheduler is responsible for placement of pods on nodes and etcd stores all of the persistent state for the cluster. Verify that all components are "Healthy".

3) `kubectl get pods --namespace=kube-system`

Kubernetes runs its system services as pods. Let's take a look at those pods. All interaction between system services is done via the API Server. Kubernetes also provides its own internal DNS server. This is used to provide domain names for...
communication between Kubernetes services. If you are wondering about the "Restarts", the cluster was stopped and restarted many times as part of the lab development. Replication controllers handle restart of these services as the lab pods get deployed.

4) kubectl get pods --namespace=kube-system -o wide

The -o wide option to get pods provides more information for you. Note that this option is available on many commands to expand the output. Try it out. Notice that you see the IP address associated with each pod. Kubernetes network architecture expects that all pods can talk to each other without NAT. There are many ways to accomplish this. In our lab we have implemented NSX-T to provide logical networking. NSX-T is a new version of NSX that implements overlay networking down to the container level.

You can also see that there are three kube-proxy pods, one running on each node. As discussed in Module 1, kube-proxy facilitates the routing of service requests across the set of pod endpoints through implementation of flow control rules. These rules are implemented in different ways depending upon the networking technology used. We have a separate lab, HOL-1826-02, that deep dives into NSX-T with Kubernetes, so we won't spend more time on that here.

**Important Note:** We have occasionally seen the kube-dns pod fail. All system pods should have a STATUS of Running. If the kube-dns pod shows CrashLoopBackoff, then execute the following command:

```
kubectl delete -n kube-system po/kube-dns-uuid
```

The uuid portion is whatever you see for the pod name in the previous command. The replication controller for this pod will automatically restart it. You can continually execute the kubectl -n kube-system get pods until you see that the kube-dns pod is running.

That's it for the system services. Let's move on to Namespaces
Namespaces and CLI context

Namespaces are intended for use in environments with many users spread across multiple teams, or projects. Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces. They are a way to divide cluster resources between multiple uses. As Kubernetes continues to evolve, namespaces will provide true multi-tenancy for your cluster. They are only partially there at this point. You can reference objects in a namespace by applying command line label/selector and permanently by setting the context for your environment. You will do both in this section.

Set Context

Before interacting with your cluster you must configure kubectl to point to your cluster and provide the namespace, along with any authentication needed. In our case, we are running the CLI on the Master node, so all we need to set up is the namespace and authentication. The following few steps will update file /home/localadmin/.kube/config to hold the kubectl configuration info. By setting up the config file, you remove the need to include that information on each kubectl command. The cluster config names the cluster and points kubectl to a specific certificate and API Server for the cluster.

Verify Config Is Correct Directly In Config File

```
localadmin@k8s-master:$ cat /home/localadmin/.kube/config
apiVersion: v1
clusters: []
contexts:
- context:
  namespace: vhubby
  name: admin@kubernetes
  current-context: admin@kubernetes
  kind: Config
  preferences: {}
  users: []
localadmin@k8s-master:$
```

The set-context command creates a config file that is used by kubectl to interact with the cluster. Our file is very simple because we are not doing any sort of trusted authentication. In production environments you might see key or certs, as well as specific user and cluster settings that explicitly define the context for how to interact with a particular cluster. In our case, we will interact with the cluster through the
vhobby namespace and have called our context admin@kubernetes. View the contents of the config file.

1) cat /home/localadmin/.kube/config

**Verify Config With kubectl**

![kubectl config view output]

You don't actually have to cat the config directly to see the configuration. kubectl provides a command to do that.

1) kubectl config view

**Namespaces**

![kubectl get namespaces output]

Let's take a look at the namespaces in our cluster. What we care about for this lab are the kube-system and vhobby namespaces. As we have previously seen, kube-system contains the Kubernetes cluster system objects. vhobby will be where we are deploying our applications.
Now we will see how the namespaces label selector changes the output of the `get` commands. Remember that our current context is vhobby, and you have not created any application pods yet. So no resources are found. The default namespace contains a single pod that is an ingress controller used by NSX-T. (For more on that and everything NSX-T with Kubernetes, try lab HOL-1826-02) Finally, as you saw previously, the kube-system namespace is running the Kubernetes cluster system services.
Deployments, Pods and Services

So far you have interacted with your Kubernetes cluster in the context of system services. You looked at pods that make up kube-system, set your CLI context and got some familiarity with CLI constructs. Now you will see how these relate to actually deploying an application. First a quick review on a couple of Kubernetes object definitions.

- **Pod** - A group of one or more containers (such as Docker containers), the shared storage for those containers, and options about how to run the containers. A pod's contents are always co-located and co-scheduled, and run in a shared context.

- **Service** - Kubernetes pods are ephemeral. When they die, they are recreated - not restarted. Replication controllers in particular create and destroy pods dynamically (e.g. when scaling up or down or when doing rolling updates). While each pod gets its own IP address, even those IP addresses cannot be relied upon to be stable over time. This leads to a problem: if some set of pods (lets call them backends) provides functionality to other pods (lets call them frontends) inside the Kubernetes cluster, how do those frontends find out and keep track of which backends are in that set? A Kubernetes Service is an abstraction which defines a logical set of pods and a policy by which to access them - sometimes called a micro-service. The set of pods targeted by a Service is (usually) determined by a Label Selector. Not only does a service provide discovery of the underlying pods, but handles East/West Load Balancing across them through the Kube-Proxy process running on each Node.

- **Deployment** - Provides declarative updates for pods and replica sets (the next-generation replication controller). You only need to describe the desired state in a deployment object, and the deployment controller will change the actual state to the desired state at a controlled rate for you. You can define deployments to create new replica sets, or remove existing deployments and adopt all of their resources with new deployments.

Just a reminder that Module 1 of this lab goes into a more detailed explanation of these components.
Defining Desired Application State

If you are not still in the CLI and need to relaunch it:

1) Click on Putty

2) Click on K8 Master

3) Click Open
Central to Kubernetes are the process control loops that attempt to continuously reconcile the actual state of the system with the desired state. The desired state is defined in object specifications that can be presented to the system from yaml or json specification files. You are going to deploy a simple nginx web server. The yaml file specification will create a Deployment with a set of pods and a service. Let's see how that works.

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: hol-nginx
  labels:
    app: hol-nginx
spec:
  replicas: 2
  template:
    metadata:
      labels:
        run: hol-nginx
    spec:
      containers:
      - name: hol-nginx
        image: harbor-01a.corp.local/library/hol-nginx:v1
        ports:
        - containerPort: 80

apiVersion: v1
kind: Service
metadata:
  name: hol-nginx
  labels:
    run: hol-nginx
spec:
  type: NodePort
  ports:
  - port: 80
    protocol: TCP
    name: http
    selector:
      run: hol-nginx
```

1) cd /home/localadmin/vhobby
Let's break apart the components of this file. Every specification includes the version of the API to use. The first spec is the deployment, which includes the "PodSpec" and replica set.

1) The deployment name is hol-nginx.

2) Notice that it has a Label, app: hol-nginx. Labels are key:value pairs that are used to specify identifying attributes of objects and are used extensively in Kubernetes for grouping. You will see one example with the service creation in the following steps.

3) Replicas specifies the desired state for the number of pods defined in the spec section that should be running at one time. In this case, 2 pods will be started. (Note: the scheduler will attempt to place them on separate nodes for availability but its best effort)

4) The pods also get their own label. This is used for, among other things, service Endpoint discovery

5) This pod is made up of a single container that will be instantiated based on the hol-nginx:V1 image stored in the harbor-01a.corp.local registry
6) The container will expose port 80. Note that this is the container port, not the host port that provides external access to the container. More on that in a minute.

---
apiVersion: v1
type: Service
name: hol-nginx
ports:
- port: 80
  protocol: TCP
selector:
  run: hol-nginx
---

The next spec is for the service. In addition to the name and label, the spec itself has two very important components:

1) Type: Nodeport By specifying nodeport, Kubernetes will expose a randomly generated port on each node in the cluster. The service can be accessed from outside the cluster via the IP address of any node combined with this port number. Access to services internal to the cluster - like a frontend webserver trying to update a backend database are done via a clusterIp and/or internal DNS name. The internal DNS name is based on the name defined for this service.

2) Selector: run: hol-nginx This is the label that the service uses to find the pods that it routes to.

**Deploy nginx Application**

```
localadmin@k8s-master:~$ cd /home/localadmin/vhobby/
localadmin@k8s-master:~$ kubectl create -f nginx.yaml
deployment "hol-nginx" created
localadmin@k8s-master:~$ kubectl get deployment
 Name  Desired  Current  Up-to-Date  Available  Age
hol-nginx  2      2        2           2        17s
localadmin@k8s-master:~$ kubectl get pods
 Name                 READY STATUS STARTS AGE
hol-nginx-2473466663-23mG0 1/1 Running 0 25s
hol-nginx-2473466663-k18vw 1/1 Running 0 25s
localadmin@k8s-master:~$ kubectl
```

The nginx.yaml defines the desired state for the deployment of this application, but we haven't defined what it actually does. nginx is an application that can act as a Web
Server or reverse proxy server. You will deploy the application, look at its running components and verify that the web server is running through your browser.

If you are not already in /home/localadmin/vhobby directory then type

1) cd /home/localadmin/vhobby

2) kubectl create -f nginx.yaml

3) kubectl get deployment

Notice that the hol-nginx deployment has a desired state of two pods and the current state is two running pods

4) kubectl get pods

Notice that you have two running pods. Try the -o wide option to see which nodes they are on and their internal IP address

**View the Service for hol-nginx**

We have running pods, but no way to access the service from our network. Remember that the pod IP addresses are private to the cluster (actually we break that rule because of the lab setup, generally this will be true). Also, what happens if the replication controller has to restart one of them and the IP changes. So we need the service to discover our application endpoints

```
localadmin@k8s-master:~/.vhobby$ kubectl get svc
NAME           CLUSTER-IP     EXTERNAL-IP   PORT(S)         AGE
hol-nginx      10.98.111.39  <nodes>       80:31025/TCP   8m

localadmin@k8s-master:~/.vhobby$ kubectl get svc
```

1) kubectl get svc

Notice that the Service has a clusterIP. This is an internal IP. Generally you would not be able to access the service through this IP. If you are on a platform that has configured a load balancer service (Like AWS Elastic LB) you would see an external IP that allows you to access that LB and be routed to your service endpoints.

Find the nodport, you will use it to access the nginx webserver. In our example the randomly generated nodeport is 31025. Remember that when a service is defined as
Type: NodePort, a randomly generated port is opened on each cluster node for access to that service. You could choose any cluster node to access the service. We are going to use the Master VM. Its IP is 10.0.1.10.

**Access nginx Web Server**

1) Click on Google Chrome

2) Enter [http://10.0.1.10:YourNodeport](http://10.0.1.10:YourNodeport)

If you see the "Welcome to Hands on Labs" Your Web Server is running.
Back to the CLI

If you closed your CLI then

```
Using username "localadmin".
Authenticating with public key "controlcenter" from agent
Welcome to Ubuntu 18.04.1 LTS (GNU/Linux 4.4.0-53-generic x86_64)

* Documentation: https://help.ubuntu.com
* Management: https://landscape.canonical.com
* Support: https://ubuntu.com/advantage

95 packages can be updated.
0 updates are security updates.

localadmin@k8s-master:~$ cd vhobby
localadmin@k8s-master:/vhobby$ [your CLI command here]
```
1) Click on Putty
2) Select K8 Master
3) Click Open
4) cd /home/localadmin/vhobby

**Replica Sets and Labels**

As discussed previously with services, the labels are very important for Kubernetes to group objects. Let's see how that works with replica sets.

1) kubectl get rs -o wide
2) kubectl get pods -l run=hol-nginx

As discussed previously with services, the labels are very important for Kubernetes to group objects. Let's see how that works with replica sets.

1) `kubectl get rs -o wide`
2) `kubectl get pods -l run=hol-nginx`

Notice the selector is based on the run=hol-nginx label. So pods with that label are monitored for restart based on this replica set.
Scale our Application Up

Applications may need to be scaled up or down to improve performance or availability. Kubernetes can do that with no application downtime by adding or removing pods. Remember that the success of scaling is dependent upon the underlying application's ability to support it. Let's scale our deployment and see what happens. Remember that scaling is changing the desired state for our app, and the replication controller will notice a difference between desired state and current state, then add replicas.

1)  kubectl scale deployment hol-nginx --replicas 4
2)  kubectl get pods

You may have to execute get pods more than once to see the new running pods, but you have gone from an application that had two copies of the nginx web server running to four replicas. The service automatically knows about the new endpoints and kube-proxy has updating the control flows to provide internal load balancing across the new pods. Pretty cool!!

Scale our Application Back Down

You can also remove unneeded capacity by reducing the number of replicas in your deployment.

1)  kubectl scale deployment hol-nginx --replicas 2
Delete Our Application

Now let's delete our deployment. It's very simple. Just reference the same spec file you used to create the deployment.

```bash
localadmin@k8s-master:~/$ hobby$ kubectl delete -f nginx.yaml
deployment "hol-nginx" deleted
service "hol-nginx" deleted
localadmin@k8s-master:~/$
```

1) `kubectl delete -f nginx.yaml`
Private Registry With Harbor

The application deployments in this lab make use of a private container registry. We are using software from a VMware opensource project called Harbor as our registry. In this section, you will take a quick look at the images you previously used in the nginx deployment and the other application images you will use in Module 3 of the lab. Most organizations will use a private registry rather than public Docker hub to improve security and latency for their applications. Harbor is discussed in more detail in Module 1 of this lab and in lab HOL-1830. Although Harbor can be deployed as a highly available application, we have not done that for this lab. The nature of these lab pods is that infrastructure can have unpredictable latency. Harbor seems to be impacted by that. If you have any trouble using the Harbor UI, we have provided remedial steps below.
Login to Harbor UI

1) Click on Google Chrome

2) Click on Harbor-01a.corp.local bookmark

3) Did you get the Harbor UI or this page?
If you see the page displayed above (or a Bad Gateway Error), execute the following steps to bring the UI back up.

1) Click on Putty

2) Select harbor-01a.corp.local
3) Click Open and login as root

```
root@harbor-01a [ ~ ]# cd harbor
root@harbor-01a [ ~/harbor ]# docker-compose down
Stopping harbor-jobservice ... done
Stopping nginx ... done
Stopping harbor-ui ... done
Stopping harbor-db ... done
Stopping harbor-adminserver ... done
Stopping harbor-log ... done
Removing harbor-jobservice ... done
Removing nginx ... done
Removing harbor-ui ... done
Removing harbor-db ... done
Removing registry ... done
Removing harbor-adminserver ... done
Removing harbor-log ... done
Removing network harbor_harbor
```

```
root@harbor-01a [ ~/harbor ]# docker-compose up -d
Creating network "harbor_harbor" with the default driver
Creating harbor-log
Creating harbor-db
Creating harbor-adminserver
Creating registry
Creating harbor-ui
Creating nginx
Creating harbor-jobservice
```

1) cd harbor

2) docker-compose down
3) docker-compose up -d

1) Reload the browser screen

Note: In one instance we found that this did not fix the problem. If this is the case for you, from the Harbor Command line:

run the command: systemctl restart docker and then reload the browser
Enter Harbor Username/Password

1) Login to Harbor with username: admin and password: VMware1!

View Projects and Repos

Harbor organizes images into a set of projects and repositories within those projects. Repositories can have one or more images associated with them. Projects can have RBAC (Role Based Access Control) and replication policies associated with them so that administrators can regulate access to images and create image distribution pipelines across registries that might be geographically dispersed. You should now be at a summary screen that shows all of the projects in this registry. There is only a single project called library.

![Harbor Interface Screenshot]

The library project contains four repositories and has no access control. It is available to the public.

1) Click on library to see the repos
You now see four different repos. The first three will be used in Module 3 to deploy our vhobby application. We used the nginx repo for the work you did in Module 2. Note that the vhobby image has two tags. This is because we have uploaded two version of that image. More on that in Module 3.

1) Click on the library/vhobby repo

Notice that there are two images. During lab preparation two versions of the same image were uploaded so that we could upgrade our application in Module 3.
That's it for Harbor and Module 2. Continue on to Module 3 for more on application deployment and management.
Conclusion

You have now become familiar with deploying a simple application on Kubernetes and using the various system constructs. You should feel comfortable with the kubectl CLI and be ready to deploy a more complex application in Module 3.

You've finished Module 2

Congratulations on completing Module 2.

Proceed to any module below which interests you most.

- **Module 1 - Introduction to Kubernetes** (30 minutes) (Advanced) you will learn what container orchestration with Kubernetes is all about. Terminology is a barrier to understanding technology and the fundamental components of the Kubernetes platform will be explained in detail. This module contains no lab steps but will provide you with a foundational understanding that will make the other modules more intuitive.
- **Module 3 - Deploy and Manage Multi-tiered Application** (30 minutes) (Advanced) This module can be taken independently from the first two, but builds on the knowledge gained there to deploy a more complex application. You will see the components of a web-based Hobby shop product rating system application that includes a backend database. Once your application is deployed, you will upgrade...
to a newer version with no downtime, and will subsequently roll the upgrade back - again without downtime.

**How to End Lab**

To end your lab click on the **END** button.
Module 3 - Deploy and Manage a Multi-Tiered Application (30 minutes)
Introduction

In this module you are going to deploy an application called Hobbyshop. The application consists of a web frontend and backend database. The database is a Redis key value store and has a Master/Slave architecture. You will have separate deployments for each of the three tiers. There will also be services associated with each deployment to provide service discovery and East/West load balancing. As part of lab setup, container images have been built for you. As an appendix to this module, we have provided the steps to do this. You are welcome to try that or you can take lab HOL-1830-01 to dive into Docker in detail.

This diagram represents the application we are going to manage. The application consists of a frontend Web Server and a Redis Key Value store. The Redis store is implemented as a single Master with multiple workers. There are three separate deployments: frontend, Redis Master and Redis Worker. Each deployment defines a replica set for the underlying pods. Persistent volumes are shown as attached to the frontend pods, however this is an example of what is possible but is not part of this lab.
Deploy and Upgrade Hobby Shop Review Application

Login to CLI

1) Click on Putty Icon

2) Select k8 Master

3) Click Open
View the Yaml Files

```yaml
localadmin@k8s-master:/vhobby$ cat frontend-deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: frontend
  # these labels can be applied automatically
  # from the labels in the pod template if not set
  # labels:
  #   app: guestbook
  #   tier: frontend
spec:
  # this replicas value is default
  # modify it according to your case
  replicas: 3
  # selector can be applied automatically
  # from the labels in the pod template if not set
  # selector:
  #   matchLabels:
  #     app: guestbook
  #     tier: frontend
template:
  metadata:
    labels:
      app: guestbook
tier: frontend
spec:
  containers:
  - name: php-redis
    image: harbor-01a.corp.local/library/vhobby:v1
    resources:
      limits:
        cpu: 100m
        memory: 100Mi
    env:
      - name: GET_HOSTS_FROM
        value: dns
        # If your cluster config does not include a dns service, then to
        # instead access environment variables to find service host
        # info, comment out the 'value: dns' line above, and uncomment the
        # line below.
        # value: env
    ports:
      - containerPort: 80
```

In Module 2 we went through the details of the deployment, pod and service specs so we won't do that again here. Let's look at our frontend deployment and see a couple of new elements.

1) cd /home/localadmin/vhobby
The only new elements from the previous yamls we viewed are the resource constraints put on the containers and the env: section which allows environment variables to be set in the container when run. Also notice that the image is vhobby:V1.

**Deploy Hobbyshop V1 Master Pod**

```
localadmin@k8s-master:/vhobby$ kubectl create -f redis-master-deployment.yaml

deployment "redis-master" created
```

```
localadmin@k8s-master:/vhobby$ kubectl get pods
NAME              READY   STATUS    AGE
redis-master-1506642955-9s178  1/1      Running   3m
```

```
localadmin@k8s-master:/vhobby$ kubectl get deployments
NAME         DESIRED CURRENT    UP-TO-DATE AVAILABLE   AGE
redis-master  1               1               1             3m
```

Now you can deploy your application. This is done using the kubectl create command and pointing to the appropriate yaml configuration files. It's important to note that we have set this up as 6 separate configuration files so that it is easier to understand what is being done. The entire application could have been deployed with a configuration file and a single kubectl create command.

1) kubectl create -f redis-master-deployment.yaml

This command deploys the Redis Master pod. It will take a minute or so to come up. Verify its running by executing:

2) kubectl get pods

View your deployment

3) kubectl get deployments

view the number of replicas for this pod. It will only be one.
4) `kubectl get rs`

```bash
localadmin@k8s-master:/vhome$ kubectl describe pods redis-master
Name:   redis-master-1506842955-8s170
Namespace: k8s-service
Node:   k8s-node/10.0.1.11
Labels: app=redis
        pod-template-hash=1506842955
        role=master
tier=backend
Annotations: Kubernetes.io/created-by= reconciliation:
             "kind":"SerializedReference","apiVersion":"
             "v1","reference":{"kind":"ReplicaSet","namespace":
             "k8s-service","name":"redis-master-1506842955","uid":"3225556e-60a3-11e7-ad89-00505688c01c"
Status: Running
IP:     10.4.0.162
Containers:
  master:
    Container ID: docker/k8s-17ad89556e-60a3-11e7-ad89-00505688c01c
    Image:   redis:image:v1
    Image ID: docker-pullable:pci.io/google_containers/redis:ssha256:2e06ebc256497f8b559b7f0789cbb10a35c0eaf2a42f77dc06b7f
    Port:    6379/TCP
    State: Running
    Ready: True
    Restart Count: 0
    Requests:
      cpu: 100m
    Memory: 150Mi
    Environment: <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-jp9j4 (ro)
Conditions:
  Type      Status     Initialized  Ready  PodScheduled
  Type:     Status
  Initialized: True
  Ready: True
  PodScheduled: True
Volumes:
  default-token-jp9j4:
    Type: Secret (a volume populated by a Secret)
    SecretName: default-token-jp9j4
    Optional: false
    QoS Class: Burstable
    Node-Selectors: <none>
    Tolerations: node.alpha.kubernetes.io/not-ready= Exists:NoExecute for 300s
                node.alpha.kubernetes.io/unreachable= Exists:NoExecute for 300s
```

For details on your pod, you can describe it

1) `kubectl describe pods redis-master`

**Deploy Hobbyshop V1 Master Service**

```bash
localadmin@k8s-master:/vhome$ kubectl create -f redis-master-service.yaml
service "redis-master" created
localadmin@k8s-master:/vhome$ kubectl get svc
NAME        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
redis-master 10.109.220.48  <none>        6379/TCP    7s
```

You will now deploy the service for the master pod. Remember that a service is an abstraction for a set of pods. It provides an endpoint for the underlying pods and load balances across the pods in a replica set.
1) kubectl create -f redis-master-service.yaml

2) kubectl get svc

Notice that there is no NodePort on the Master service. That is because it is only accessed from inside the cluster and does not need ingress from an external network. Compare that with the frontend we will deploy in a few steps.

**Deploy Hobbyshop V1 Worker Pod and Service**

You will repeat the deployment and service creation for the Worker deployment and service:

1) kubectl create -f redis-slave-deployment.yaml

2) kubectl create -f redis-slave-service.yaml

3) kubectl get svc

**Deploy Hobbyshop V1 Frontend Webpage Pod and Service**

1) kubectl create -f frontend-deployment.yaml

1 2 3

---

Notice that there is no NodePort on the Master service. That is because it is only accessed from inside the cluster and does not need ingress from an external network. Compare that with the frontend we will deploy in a few steps.

**Deploy Hobbyshop V1 Worker Pod and Service**

You will repeat the deployment and service creation for the Worker deployment and service:

1) kubectl create -f redis-slave-deployment.yaml

2) kubectl create -f redis-slave-service.yaml

3) kubectl get svc

**Deploy Hobbyshop V1 Frontend Webpage Pod and Service**

1) kubectl create -f frontend-deployment.yaml
2) `kubectl get pods -o wide`

```bash
localadmin@k8s-master:~/v hobby$ kubectl create -f frontend-service-nodeport.yaml
service "frontend" created
localadmin@k8s-master:~/v hobby$
```

3) `kubectl create -f frontend-service-nodeport.yaml`

```bash
localadmin@k8s-master:~/v hobby$ kubectl describe svc frontend
Name:       frontend
Namespace:  v hobby
Labels:     app=guestbook
tier=frontend
Annotations: <none>
Selector:   app=guestbook,tier=frontend
Type:       NodePort
IP:         10.109.46.141
Port:       http  80/TCP
            http  30298/TCP
Endpoints:  10.4.0.165:80, 10.4.0.166:80, 10.4.0.167:80
Session Affinity: None
Events:     <none>
localadmin@k8s-master:~/v hobby$
```

4) `kubectl describe svc frontend`

Notice the Nodeport value on the frontend Service. This is the port you will use to access the webserver on the browser. Remember it for the next step. Also the endpoints are the internal IPs for the Pods that the service Load Balances across.
Access Hobby Shop Review Webpage.
Access the Hobbyshop application from your browser. This process isn’t intuitive. Kubernetes exposes a Nodeport on all Worker/Master nodes. The service uses kube-proxy to forward connection on that Nodeport to one of the running containers. We will connect to the frontend service through the Master and our connection will be proxied to one of the containers. Since NodePort is exposed on every node in the Kubernetes cluster, you could choose any node IP to connect with. We are going to use the master. The Master IP is 10.0.1.10. You can find this by entering "ip a" on the command line and looking for the ens160 interface.
1) Click on Google Chrome

2) Enter the MasterIP:NodePort in the browser. In my example this will be 10.0.1.10:30298
Enter Review in the Application

The Hobby Shop Review application lets you enter a review for a single product on the home screen.

Virtual Hobbyshop V1

Extra 330SC Radio-Controlled Airplane - An excellent aerobatic plane for the intermediate to advanced pilot. Comes fully loaded with an electric motor, speed controller and servos. This plane will excel at traditional aerobatic as well as 3D extreme aerobatics. Just add radio and batteries. A great value!

Requires: A 6 channel radio system and 4s LiPo 3000-3700 mAh

Stock status: In stock

Shipping: Ships in 2-3 day

Rate and review this product

Customer Feedback

By demcuser

By demcuser Love This Product!!!
1) Click on the number of Stars you want to give this product

2) Write a short review of your experience

3) Click Submit

Notice that the information was saved to your Redis database and then read back out and displayed on the page. Also take note of the title: Hobby Shop V1

Note: if your review did not get retrieved from Redis and printed on screen, it is probably because the kube-dns service failed. We have seen that occasionally in this lab. The steps to fix that are as follows:

Only do this step if your review did not print correctly

```
kubectl get pods --namespace=kube-system

kubectl delete -n kube-system po/kube-dns-uuid
```

The uuid portion is whatever you see for the pod name in the previous command. The replication controller for this pod will automatically restart it. You can continually execute the kubectl -n kube-system get pods until you see that the kube-dns pod is running.
Upgrade Hobby Shop Application

Now we will see how quickly and easily you are able to rollout a new version of this app without any downtime. Kubernetes will simply create new pods with a new upgrade image and begin to terminate the pods with the old version. The service will continue to load balance across the pods that are available to run.

```
localadmin@k8s-master:/vhobby$ cat frontend-deployment-V2.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: frontend
# these labels can be applied automatically
# from the labels in the pod template if not set
labels:
  # app: guestbook
  # tier: frontend
spec:
  # this replicas value is default
  # modify it according to your case
  replicas: 3
  # selector can be applied automatically
  # from the labels in the pod template if not set
  selector:
    matchLabels:
      # app: guestbook
      # tier: frontend
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: harbor-01a.corp.local/library/vhobby:V2
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
          - name: GET_HOSTS_FROM
            value: dns
            # If your cluster config does not include a dns service, then to
            # instead access environment variables to find service host
            # info, comment out the 'value: dns' line above, and uncomment the
            # line below.
            value: env
        ports:
          - containerPort: 80
```

1) From the CLI- cat frontend-deployment-V2.yaml
2) Notice that the image changed to vhobby:V2

```bash
localadmin@k8s-master:/vhobby$ kubectl apply --record=true -f frontend-deployment-V2.yaml
Warning: kubectl apply should be used on resource created by either kubectl create --save-config or kubectl apply deployment "frontend" configured

localadmin@k8s-master:/vhobby$ kubectl get pods
```

1) `kubectl apply --record=true -f frontend-deployment-V2.yaml`

2) `kubectl get pods`
You should see new pods creating and old terminating, but it happens fast.

1) Click on your Chrome Browser

2) Refresh The Page and notice that the image is V2 and that your Review is still there.
Roll Back Hobby Shop Application Upgrade

Uh oh!! Users aren't happy with our application upgrade and the decision has been made to roll it back. Downtime and manual configuration, right? Nope. It's a simple reverse of the upgrade process.

```
localadmin@k8s-master:/vhooby$ kubectl rollout history deployment/frontend
REVISION CHANGE-CAUSE
1 <nco:notyourimage>
2 kubectl apply --record=true --filename=frontend-deployment-V2.yaml

localadmin@k8s-master:/vhooby$ kubectl rollout undo deployment/frontend --to-revision 1
localadmin@k8s-master:/vhooby$ kubectl get pods
```

1) `kubectl rollout history deployment/frontend`

Notice that you have change tracking across all of your deployment revisions. In our case we have made only one change. So we will roll back to our original image.

2) `kubectl rollout undo deployment/frontend --to-revision 1`

3) `kubectl get pods`
You should see terminating pods and new pods creating.

Once they are all running, Go back to chrome and refresh the Browser again. You should see V1

**Appendix - Build and Push Container Images to Harbor**

This is a supplemental section to Module 3. If you are interested in the details of building images and pushing them to a registry, try this out. It requires that you have a Harbor registry that is running correctly. If you have issues with Harbor, see the remedial steps in the Harbor section of Module 2. The short of it is you must run docker-compose down and docker-compose up -d from the /root/harbor directory on the harbor-01a.corp.local VM
This section will walk you through how to build the frontend image for the vhobby application, tag it appropriately and push to Harbor.

For a more comprehensive look at working with Docker and vSphere Integrated Containers, try labs HOL-1830-01 and HOL-1830-02.

**Login to Harbor VM**

1. Click on Putty Icon
2. Select harbor-01a.corp.local
3. Click Open
4. Log in as: root
   
   Authenticating with public key "controlcenter" from agent
   
   
   root@harbor-01a [ ~ ]#
3) select open

4) username is root

**Change to Source Code Directory**

```
root@harbor-01a [ ~ ]# cd /root/vhobby/
root@harbor-01a [ ~/vhobby ]#  
```

cd /root/vhobby
Build your Docker image from the source files and verify that its in the images list. This will be very quick because all of the dependent layers are cached locally. What you are seeing on the screen are each of the commands from the Dockerfile in that directory. This file specifies how to build the image. (Don't forget the space . at the end of this command) The image is stored in the local docker image cache.

1) docker build -t hobbyshop:V3 .

2) docker images
Tag Images

You must tag this image with the local Registry FQDN, so that Docker knows where to push them. Docker images command will show you the tagged image.

1) docker tag hobbyshop:V3 harbor-01a.corp.local/library/vhobby:V3
2) docker images

Notice that the image ids are the same for V2 and V3. That is because we did not make changes in the code before building the image. You could edit the index.html file and then build again if you want to see a different image.
Login To The Registry and Push Images

1) docker login -u admin -p VMware1! http://harbor-01a.corp.local
   Login Succeeded
2) docker push harbor-01a.corp.local/library/vhobby:V3
   The push refers to a repository [harbor-01a.corp.local/library/vhobby]
   197470f0b4a6 : Layer already exists
   71263770c31f : Layer already exists
   a25f88a2f01f : Layer already exists
   75e14d7090c9 : Layer already exists
   94d9ea7e1f69 : Layer already exists
   e6e6b2900332 : Layer already exists
   e7595de33e43 : Layer already exists
   704bf50245a : Layer already exists
   899ac7763988 : Layer already exists
   2fdbc3009d35 : Layer already exists
   20c1acbeced8 : Layer already exists
   67c33e87308d : Layer already exists
   3470f1a5762 : Layer already exists
   37b36370b00a : Layer already exists
   62e68c33e872 : Layer already exists
   dda8eb95fffd : Layer already exists
   e42e45043c0b : Layer already exists
   8103b6ed3d4 : Layer already exists
   ac746576a1 : Layer already exists
   753f5b581a4a : Layer already exists
   98ac6c32b9b7 : Layer already exists
   8c31ea775f91 : Layer already exists
   955c46160919 : Layer already exists
   c40f634e46bb : Layer already exists
   009e0f1d4bba : Layer already exists
   V3: digest: sha256:a5f66fdebb4a5291c93aba2a2b11a45da5e3420cd610f52d2d8048f4a5c020688 size: 5518

1) docker login -u admin -p VMware1! http://harbor-01a.corp.local
2) docker push harbor-01a.corp.local/library/vhobby:V3

This image is now available to be used for container deployment.
Verify Image through Harbor UI

1) Click on Google Chrome
2) Click on harbor-01a.corp.local bookmark

3) login with Username: admin Password: VMware1!

<table>
<thead>
<tr>
<th>Project Name</th>
<th>Access Level</th>
<th>Role</th>
<th>Repositories Count</th>
<th>Creation Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>library</td>
<td>Public</td>
<td>Project Admin</td>
<td>4</td>
<td>7/17/2017, 6:27 AM</td>
</tr>
</tbody>
</table>

1) Click on Library Project

2) Click on vhobby Repo and verify that your V3 image is there

You have now completed Module 3 and the Kubernetes Basics Lab
Conclusion

You have now deployed a multi-tier application using Kubernetes and have rolled out an upgrade to that application without and downtime. You also saw that you could easily roll back to a previous version, also without downtime. If you have taken all three Modules, this concludes the Kubernetes Basics Lab.

You've finished Module 3

Congratulations on completing Module 3.

Proceed to any module below which interests you most.

- **Module 1 - Introduction to Kubernetes** (30 minutes) (Advanced) you will learn what container orchestration with Kubernetes is all about. Terminology is a barrier to understanding technology and the fundamental components of the Kubernetes platform will be explained in detail. This module contains no lab steps but will provide you with a foundational understanding that will make the other modules more intuitive.

- **Module 2 - Kubernetes Deep Dive** (45 minutes) (Advanced) This module is pure Kubernetes lab work. You will use the kubectl CLI to manipulate the Kubernetes components you read about in Module 1. You will create the deployment that defines the pods and services that make up your nginx webservice application. You will scale the application both up and down - adding
pods to provide either increased capacity or availability. By the end you should be very comfortable with manipulating your application components through the CLI.

How to End Lab

To end your lab click on the END button.
Conclusion

Thank you for participating in the VMware Hands-on Labs. Be sure to visit http://hol.vmware.com/ to continue your lab experience online.

Lab SKU: HOL-1831-01-CNA

Version: 20180215-205252