## Table of Contents

Lab Overview - HOL-1832-01-CNA - Pivotal Container Service (PKS) and Kubernetes - Getting Started ................................................................................................................................. 2  
Lab Guidance ............................................................................................................................................... 3  

### Module 1 - Introduction to Kubernetes (45 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 - Introduction to Pivotal Container Service (PKS) (45 minutes) (Advanced) ... 28  
  Introduction ................................................................................................................................................ 29  
  Our Lab Environment and Deployed Kubernetes Cluster ......................................................... 32  
  Cluster Scale, Health Monitoring and Troubleshooting ............................................................. 40  
  Persistent Volumes and Kubernetes Storage Policy ................................................................. 48  
  NSX Network and Security Policy .......................................................................................... 53  
  Harbor Enterprise Container Registry ................................................................................... 66  
  Conclusion ................................................................................................................................................. 82  

### Module 3 - Kubernetes Deep Dive (45 minutes) .................................................................... 84  
  Your Lab Kubernetes Cluster ............................................................................................................. 85  
  Namespaces and CLI context ............................................................................................................. 90  
  Deployments, Pods and Services ............................................................................................... 93  
  Conclusion ................................................................................................................................................. 102  

### Module 4 - Deploy and Manage a Multi-Tiered Application (30 minutes) ......................... 104  
  Introduction .............................................................................................................................................. 105  
  Deploy and Upgrade Restaurant Review Application to Add Persistent Volumes 107  
  Conclusion ................................................................................................................................................. 123
Lab Overview - 
HOL-1832-01-CNA - 
Pivotal Container Service (PKS) and 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 be introduced to Pivotal Container Service (PKS), a purpose built service for operationalizing Kubernetes at scale. 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** (45 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. If you already have have an understanding of Kubernetes and want to get hands-on, start with Module 2.

- **Module 2 - Introduction to Pivotal Container Service (PKS)** (45 minutes) (Advanced) Operationalizing Kubernetes at scale is not for the faint of heart. PKS is a purpose built service to deploy and manage Kubernetes clusters. This module will introduce PKS, showing the capabilities to extend High Availability, scale, health monitoring and lifecycle management to Kubernetes.

- **Module 3 - 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 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 4 - 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 restaurant 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**

![Image showing keyboard layout]

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**

![Image showing keyboard layout]

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 (45 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 and Pivotal Container Service (PKS).
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 and Pivotal Container Service (PKS).

**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**

Each Blue Box is a VM and represents a 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**

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. 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. In our lab we are using NSX to provide the networking. NSX provides the capability to define a Load Balancer which will proxy access to the underlying Services.

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. NSX provides load balancing for clusters deployed through Pivotal Container Service (PKS).
- 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 Restaurant Rating Application

This simple application captures votes for a set of restaurants, provides a running graphical tally and captures the number of page views. It contains four separate deployments—UI, Application Server, Postgres DB and Redis caching server. A deployment provides a declarative method for defining pods, replica sets and other Kubernetes constructs. The UI Deployment includes a UI pod, which runs an Nginx Webserver. It defines a replica set that maintains three running copies of the UI pod. It also defines a UI service that provides an abstraction to the underlying UI pods, including a ClusterIP and Load Balancer that can be used to access the service.

The application is using a Redis Key:Value store to capture page views and a Postgres Database to persist the vote. 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 UI deployment and the UI service. The other yaml files are available as part of module 4.

```
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: yelb-ui
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: yelb-ui
tier: frontend
    spec:
      containers:
      - name: yelb-ui
        image: harbor.corp.local/library/restview-ui:v1
        ports:
        - containerPort: 80
```

- **apiVersion**: What kind of object is being defined
- **kind**: Give it a name
- **metadata**: Run one copy of the pod
- **spec**: Labels and Selectors
- **containers**: container image
- **ports**: 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 - yelb-ui. It defines a replica set that includes 1 replica. That means the desired state for this deployment is that 1 copy of the 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 harbor.corp.local/library/restreview-ui:V1 image. Resources can be constrained for the container based on the requests Key. 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.

```
apiVersion: v1
kind: Service
metadata:
  name: yelb-ui
  labels:
    app: yelb-ui
    tier: frontend
spec:
  type: LoadBalancer
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: yelb-ui
    tier: frontend
```

This file defines the UI service specification. The important pieces are the Type: LoadBalancer and the Selector. Specifying Type: LoadBalancer means that NSX will associate a Load Balancer with this service to provide external access to the application. The service will then route requests to one of the pods that has a label from the service's selector. So all pods with matching labels will be included in this service. Note: NSX actually changes the routing mechanism from what is described here, but it logically works this way.
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

---

**Kubernetes Worker Nodes**

- **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 see how to deploy and manage your clusters with Pivotal Container Service (PKS).
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 - Introduction to Pivotal Container Service (PKS)** (45 minutes) (Advanced) Operationalizing Kubernetes at scale is not for the faint of heart. PKS is a purpose built service to deploy and manage Kubernetes clusters. This module will introduce PKS, showing the capabilities to extend High Availability, scale, health monitoring and lifecycle management to Kubernetes.

- **Module 3 - 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 4 - 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 restaurant 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 2 - Introduction to Pivotal Container Service (PKS) (45 minutes) (Advanced)
Introduction
In this module you will see how to operationalize Kubernetes through Pivotal Container Service (PKS). What does that mean? Let’s start by looking at what Kubernetes does well. It allows developers to easily deploy applications at scale. It handles the scheduling of workloads (via pods) across a set of infrastructure nodes. It provides an easy to use mechanism to increase availability and scale by allowing multiple replicas of application pods, while monitoring those replicas to ensure that the desired state (number of replicas) and the actual state of the application coincide. Kubernetes also facilitates reduced application downtime through rolling upgrades of application pods. PKS is providing similar capabilities for the Kubernetes platform itself. Platform engineering teams are becoming tasked with providing a Kubernetes "Dialtone" service for their development teams. Kubernetes is not a simple platform to manage, so the challenge becomes how to accomplish this without architect level knowledge of the platform. Through PKS, platform engineering teams can deliver Kubernetes clusters through a single API call or CLI command. Health monitoring is built into the platform, so if a service fails or a VM crashes, PKS detects that outage and rebuilds the cluster. As resources become constrained, clusters can be scaled out to relieve the pressure. Upgrading Kubernetes is not as easy as upgrading the application pods running on the cluster. PKS provides rolling upgrades to the Kubernetes cluster itself. The platform is integrated with the vSphere ecosystem, so platform engineers can use the tools they are familiar with to manage these new environments. Lastly, PKS includes licensed and supported Kubernetes, Harbor Enterprise Container Registry and NSX-T - and is available on vSphere and public cloud platforms.
That last paragraph sounded like a marketing message, so let's net this out. PKS gives you the latest version of Kubernetes - we have committed to constant compatibility with Google Container Engine (GKE), so you can always be up to date - an easy to consume interface for deploying Kubernetes clusters, scale out capability, Health Monitoring and automated remediation, Rolling upgrade, enterprise container registry with Notary image signing and Clair vulnerability scanning. All of this deployed while leveraging NSX-T logical networking from the VMs down to the Kubernetes pods. Let's jump in.
Our Lab Environment and Deployed Kubernetes Cluster

In this lab we have done the PKS installation and deployed a small Kubernetes cluster for you to work on. Because of latency concerns in the nested infrastructure we use for these labs, we try not to create VMs as part of the lab. The Kubernetes cluster in your lab took about 8 minutes to deploy and be up and running. We will start by looking at some of the components of the cluster and then PKS.

Connect to Kubernetes Dashboard

Connecting to the Kubernetes dashboard is a little confusing because of our lab environment. Without going into all of the details, we need to create an ssh tunnel from our Windows VM to a VM running the kubectl CLI. The kubectl CLI has the ability to act as a proxy for Kubernetes API calls. When we launch our browser a connection to the Kubernetes API will be proxied through our kubectl CLI. Remember that you can highlight text in the manual and drag it to the console window. You don't have to type out every command.
1. Click on **Putty**
2. Click on **Tunnel**
3. Click on **Load**
4. Click on **Open**
5. `ubuntu@ubuntu:~` Using environment '172.31.0.2' as user 'director' (bosh:*readonly, bosh:*:admin, bosh:*:openid, bosh:*:execute, bosh:*:context)

Name: p-bosh
UUID: bf86f692-8e34-4f0e-97c0-0308e1d7dc1
Version: 261.5.0 (00000000)
CRI: vsphere_cri
User: ops_manager

Succeeded `ubuntu@ubuntu:~` pks get-credentials my-cluster

Fetching credentials for cluster my-cluster.
Context set for cluster my-cluster.

You can now switch between clusters by using: `kubectl config set-context <cluster-name>

`ubuntu@ubuntu:~` kubectl proxy

Starting to serve on 127.0.0.1:8001

---

Pivotal Container Service (PKS) and Kubernetes Getting Started
5. Type `pks get-credentials my-cluster`

This will configure kubectl to point to your cluster. Note: The Kubernetes cluster gets created as part of the initial lab deployment. In rare instances, if you started with Module 2 the cluster may not have completed the creation process. You can run the command: `pks list-clusters` to determine if the create has "Succeeded" or is "In Progress". If it's in progress you may have to wait a couple of minutes for it to complete.

6. Type `kubectl proxy`

This will configure kubectl to point to your cluster

Now the kubectl CLI is listening on port 8001 and proxying requests to the kubernetes API server.

### Launch Browser and Connect to Dashboard

1. Click on `Google Chrome`
2. Enter `localhost:8001/ui`

You should now be on the Overview page of the Kubernetes dashboard. This isn't particularly useful because we aren't running any pods at this point, but let's take a look at the cluster nodes. Note: if you are having trouble connecting to the dashboard, it is possible you "Double-Clicked" on the Tunnel connection. This actually will give you the Default connection instead of Tunnel. Make sure to Load the Tunnel connection and then Open as documented above.

3. Click on `Nodes`
You see that our cluster contains two worker nodes and they are consuming very little resource at this point. Your node names will be slightly different because the unique ID is generated with each new cluster creation. Let's drill in.

<table>
<thead>
<tr>
<th>Name</th>
<th>Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>vm-494116fa-b52e-4c9f-a0b8-ef86dd44f</td>
<td>beta.kubernetes.io/arch: amd64</td>
</tr>
<tr>
<td></td>
<td>beta.kubernetes.io/os: linux</td>
</tr>
<tr>
<td></td>
<td>kubernetes.io/hostname: 172.31.2.4</td>
</tr>
<tr>
<td></td>
<td>spec.ip: 172.31.2.4</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>vm-1a87bd39-b336-4c3a-a13f-05c55219</td>
<td>beta.kubernetes.io/arch: amd64</td>
</tr>
<tr>
<td></td>
<td>beta.kubernetes.io/os: linux</td>
</tr>
<tr>
<td></td>
<td>kubernetes.io/hostname: 172.31.2.3</td>
</tr>
<tr>
<td></td>
<td>spec.ip: 172.31.2.3</td>
</tr>
</tbody>
</table>
1. Click on the second Node.

Now you can get detailed information on your Node. Take some time to move around and get familiar with the information available through the dashboard. For those of you that have been involved with Kubernetes over the last year, you can see that the dashboard has become more and more useful with each release. We are now going to focus on the PKS CLI.
Connect To PKS CLI

1. Click on **Putty**
2. Click on **cli-vm**
3. Click on **Load**
4. Click on **Open**

Login to PKS

```
ubuntu@ubuntu:/apps4  pks login -a https://10.40.14.4:9021 -u i6X_KHe1_bSREJIo60FeKbFzE8km2KXcop -p JpgwsbdJ9L-JVx6sHbq8oqDEyeQIR -k
```

API Endpoint: https://10.40.14.4:9021
User: i6X_KHe1_bSREJIo60FeKbFzE8km2KXcop
```

Remember that you can highlight text in the manual and drag it to the console window. You don't have to type out every command.

Operations Engineers can automate day 1 and day 2 operations on their Kubernetes clusters through the CLI, or by making RESTful API calls directly. The login command authenticates you to the PKS Controller API. An auth token is downloaded and stored in `/home/ubuntu/.pks/creds.yml` and will be used for future access to the API.

### Showing Kubernetes Cluster Details

1. Type: `pkgs list-clusters`
2. Type: `pkgs show-cluster my-cluster`

The PKS CLI is designed to provide Kubernetes specific abstractions to the underlying Bosh API. Bosh is an opensource project that provides IaaS, as well as day 2 operations for Cloud platforms. It is what is used by PKS to deploy and monitor Kubernetes clusters. Bosh has tremendous capability in managing many different types of applications. That capability is available through the Bosh CLI where it has not yet been surfaced through the PKS CLI.

You will see how to use some of those Bosh commands further on in the lab.
Deploy a Kubernetes Cluster (Do Not Execute)

1. Type `pkgs create-cluster my-kube -e 10.40.14.34 -n`

(This command will fail if you run it.) Due to time and resource constraints, only one cluster can be created in this environment.

The IP address (10.40.14.34) comes from a pool of routable IPs that are defined at PKS deployment. It is the endpoint API for the created cluster. Note the Plan Id: Administrators can create plans that will define the resources and configuration for the VMs that make up the cluster nodes. In this case we have taken the default Plan.
Cluster Scale, Health Monitoring and Troubleshooting

In this section we will see how PKS allows the addition of more resources to a cluster by scaling out the number of Worker nodes. We will test cluster resiliency by killing one of the nodes and dig into some Bosh commands to monitor and troubleshoot the cluster.

Scale Cluster With PKS (Do Not Execute)

PKS allows clusters to be scaled out with a single CLI command. In this lab environment we will not execute the command because of resource and time constraints. So please do not execute!!

This command will cause a new worker node VM to be provisioned and the kubelet will be registered with the kubernetes master. It becomes very easy to add resources on demand.

Health Monitoring

PKS provides monitoring of services running in the Cluster VMs, as well as the VMs themselves. Let's see what happens when we power off one of the worker nodes.

We are going to use the Bosh CLI directly to monitor this activity.
Each Kubernetes cluster that we create is considered a Bosh deployment. A detailed discussion of Bosh is beyond the scope of this lab, but it’s important to know that the PKS api is abstracting calls to the underlying Bosh api. The deployment that starts with "Service-instance is our cluster.

1. Type `bosh -e kubobosh deployments`

Now we want to see the individual instances (VMs) that make up this deployment

1. Type `bosh -e kubobosh -d service-instance_6a4b1331-ba31-4c9d-bbc9-d8604853504e instances`

Your service-instance is different from what you see in the manual, you can highlight it and right-click from the results of the previous command to paste it on the command line.

2. Note the IP address so you can find the VM in vCenter.

Notice that all of the VMs are "Running". We are going to power one of the worker nodes down.
1. Click on Google Chrome
2. Select HTML5 Client
3. Check Use Windows session authentication
4. Click Login
Find Worker VM To Power Off

1. Click **Hosts and Clusters View**
2. Expand **PKS Resource Pool**
3. Find **VM** that matches the IP
**Power Off VM**

Now we will power off the VM. Note: In this lab environment you must power off a Worker node, not the Master node. PKS supports recovering the Master but we have not set that up in this lab.

1. Right click on the **VM**, select **Power**, then **Power Off**
Monitor With Bosh

Return to the cli-vm you were using previously

1. Press the **Up Arrow key** on your keyboard to get the previous Bosh Instances command. Press Enter

Note: you can also enter the entire command: Type bosh -e kubobosh -d service-instance_6a4b1331-ba31-4c9d-bbc9-d8604853504e instances Your service-instance ID will be different. You can find it with the bosh -e kubobosh deployments command you used earlier

In a few seconds after the power off, Bosh detects that the Agent running on that VM is unresponsive. It should take about 2 minutes to restart the VM, the kubernetes services and register the kubelet with the master. You can return to vCenter and watch recent tasks to see the VM Power On and Reconfig tasks.

Find The Bosh Task

1. Press the **Up Arrow key** again and change the "Instances" command to "tasks -ar" Press Enter

Note: you can also enter the entire command: Type bosh -e kubobosh -d service-instance_6a4b1331-ba31-4c9d-bbc9-d8604853504e tasks -ar Your service-instance ID will be different. You can find it with the bosh -e kubobosh deployments command you used earlier
This command shows the Bosh Scan and Fix task that has identified the Unresponsive Agent

2. Press the **Up Arrow key** again and change "tasks -ar" to "task ID" where the ID came from the previous command. Press Enter

Note: you can also enter the entire command: Type bosh -e kubobosh -d service-instance_6a4b1331-ba31-4c9d-bbc9-d8604853504e task "ID" Your service-instance ID will be different. You can find it with the bosh -e kubobosh deployments command you used earlier

This shows step by step how Bosh is resolving the Unresponsive Agent issue.

Once the Scan and Fix task has completed (Should take a couple of minutes), you can execute the Bosh Instances command again to see the running VMs.

1. Press the **Up Arrow key** until you get back to the Bosh instances command you executed earlier and Press Enter

Note: you can also enter the entire command: Type bosh -e kubobosh -d service-instance_6a4b1331-ba31-4c9d-bbc9-d8604853504e instances Your service-instance ID will be different. You can find it with the bosh -e kubobosh deployments command you used earlier

**Additional Troubleshooting**

Bosh provides commands for ssh into the cluster VMs and capturing the Kubernetes Log files. Try them out if you have time.

1. Type `bosh -e kubobosh -d service-instance_6a4b1331-ba31-4c9d-bbc9-d8604853504e ssh worker/87333ba4-3473-4959-8f53-a35282f5f7df`
You must substitute your deployment ID and worker name if they are different from the manual. Type Exit in the VM to return to the CLI VM

2. Type `bosh -e kubobosh -d service-instance_6a4b1331-ba31-4c9d-bbc9-d8604853504e logs` This command will consolidate all of the logs from every cluster node into a single tarball. Adding a VM name to the end will return just the logs for that VM. You can find an example in /home/ubuntu/apps/logs directory
Persistent Volumes and Kubernetes Storage Policy

Although it is relatively easy to run stateless Microservices using container technology, stateful applications require slightly different treatment. There are multiple factors which need to be considered when handling persistent data using containers, such as:

- Kubernetes pods are ephemeral by nature, so the data that needs to be persisted has to survive through the restart/re-scheduling of a pod.
- When pods are re-scheduled, they can die on one host and might get scheduled on a different host. In such a case the storage should also be shifted and made available on the new host for the pod to start gracefully.
- The application should not have to worry about the volume & data. The underlying infrastructure should handle the complexity of unmounting and mounting.
- Certain applications have a strong sense of identity (e.g.; Kafka, Elastic) and the disk used by a container with certain identity is tied to it. It is important that if a pod with a certain ID gets re-scheduled for some reason then the disk associated with that ID is re-attached to the new pod instance.
- PKS leverages vSphere Storage for Kubernetes to allow Pods to use enterprise grade persistent storage.

Persistent volumes requested by stateful containerized applications can be provisioned on vSAN, iSCSI, VVol, VMFS or NFS datastores.
Kubernetes volumes are defined in Pod specifications. They reference VMDK files and these VMDK files are mounted as volumes when the container is running. When the Pod is deleted the Kubernetes volume is unmounted and the data in VMDK files persists.

PKS deploys Kubernetes clusters with the vSphere storage provider already configured. In Module 4 you will upgrade an existing application to add persistent volumes and see that even after deleting your pods and recreating them, the application data persists. In order to use Persistent Volumes (PV) the user needs to create a PersistentVolumeClaim (PVC) which is nothing but a request for PVs. A claim must specify the access mode and storage capacity, once a claim is created PV is automatically bound to this claim. Kubernetes will bind a PV to PVC based on access mode and storage capacity but a claim can also mention volume name, selectors and volume class for a better match. This design of PV-PVCs not only abstracts storage provisioning and consumption but also ensures security through access control.

Static Persistent Volumes require that a vSphere administrator manually create a (virtual disk) VMDK on a datastore, then create a Persistent Volume that abstracts the VMDK. A developer would then make use of the volume by specifying a Persistent Volume Claim.

**Dynamic Volume Provisioning**

With PV and PVCs one can only provision storage statically i.e. PVs first needs to be created before a Pod claims it. However, with the StorageClass API Kubernetes enables dynamic volume provisioning. This avoids pre-provisioning of storage and storage is provisioned automatically when a user requests it. The VMDK’s are also cleaned up when the Persistent Volume Claim is removed.
The StorageClass API object specifies a provisioner and parameters which are used to decide which volume plugin should be used and which provisioner specific parameters to configure.

**Create Storage Class**

1. Type: `cd /home/ubuntu/apps`
2. Type: `cat redis-sc.yaml`

The yaml defines the vSphere volume and the set of parameters the driver supports.

vSphere allows the following parameters:

- **diskformat** which can be thin(default), zeroedthick and eagerzeroedthick
- **datastore** is an optional field which can be VMFSDatastore or VSANDatastore. This allows the user to select the datastore to provision PV from, if not specified the default datastore from vSphere config file is used.
- **storagePolicyName** is an optional field which is the name of the SPBM policy to be applied. The newly created persistent volume will have the SPBM policy configured with it.
- **VSAN Storage Capability Parameters** (cacheReservation, diskStripes, forceProvisioning, hostFailuresToTolerate, iopsLimit and objectSpaceReservation) are supported by vSphere provisioner for vSAN storage. The persistent volume created with these parameters will have these vSAN storage capabilities configured with it.

3. Type: `kubectl apply -f redis-sc.yaml`

Let's apply this yaml to create the storage class

4. Type: `kubectl get sc`
The command shows the created storage class.

Create Persistent Volume Claim

```yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: redis-slave-claim
  annotations:
    volume.beta.kubernetes.io/storage-class: thin-disk
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
```

Dynamic provisioning involves defining a Persistent Volume Claim that refers to a storage class. Redis-slave-claim is our persistent volume claim and we are using the thin-disk storage class that we just created.

1. Type `cat redis-slave-claim.yaml`

Let's create our Persistent Volume Claim

2. Type `kubectl apply -f redis-slave-claim.yaml`

This shows that our Persistent Volume claim was created and bound to a Volume. The Volume is a vSphere VMDK. Let's look at it in more detail.

```yaml
NAME                  STATUS       VOLUME                        CAPACITY   ACCESS Modes   STORAGE Class   AGE
redis-slave-claim     Bound       pvc-572ad22e-f244-11e7-b49c-005056684acc  2Gi        RWO           thin-disk      5m
```

1. Type `kubectl get pvc`
2. Type `kubectl describe pvc redis-slave-claim`

Here you can see that the provisioning of the volume succeeded. Let's go to vCenter and see the volume.

**View The Volume in vCenter**

1. Connect to `vcenter client` and click on the `Storage icon`
2. Select your `datastore RegionA01-iSCSI01-COMP01`
3. Select the `kubevols` folder
4. Here is the Persistent Volume you just created. Note that the volumeID in the `kubectl describe` maps to the vmdk name.

Also note that it was thin provisioned based on the storage class specification we used.

You will see how to mount this volume in your pod as part of Module 4
NSX Network and Security Policy

PKS includes software defined networking with NSX. NSX supports logical networking from the Kubernetes cluster VMs to the pods themselves providing a single network management and control plane for your container based applications. This section will not be an exhaustive look at all of the NSX Kubernetes integration - for that check our lab HOL-1826-02-NET - but will focus on a few examples. Also, this section assumes some knowledge of Kubernetes, kubectl and yaml configuration files. For an intro into some of that, you might want to take modules 3 and 4 of this lab before tackling the networking and security.

Namespaces

PKS deployed clusters include an NSX system component that is watching for new namespaces to be created. When that happens, NSX creates a new Logical Switch and Logical Router, and allocates a private network for pods that will later be attached to that switch. Note that the default is to create a NAT'd network, however you can
override that when creating the namespace to specify a routed network. Let's see what happens when we create a namespace.

### Create Namespace

We will now create a new namespace and set the context so that the cli is pointed to the new namespace. Return to the cli-vm putty session you were using earlier.

1. Type `kubectl create namespace yelb-app`
2. Type `kubectl get namespace`
3. Type `kubectl config set-context my-cluster --namespace yelb-app`

This command changes the context for kubectl so that the default namespace to use is the new yelb-app. It keeps you from having to specify the namespace on each command.
View New Objects With NSX-Mgr

1. Click on **Google Chrome Browser**
2. Click on **NSX-Mgr** bookmark
3. Enter Username: **admin**  Password: **VMware1!**
4. Click **Log in**
View Logical Router Created Automatically

There are T1 routers created for each of our namespaces and the yelb-app T1 router was automatically added when we created the Namespace. If you click on Switching you would see a similar list of Logical Switches. When pods are deployed, Ports are created on the appropriate switch and an IP from the pool is assigned to the pod.

1. Click on Routing
2. Click on T1 Router created for the yelb-app namespace
Kubernetes Network Policy and Microsegmentation

Using Network Policy, users can define firewall rules to allow traffic into a Namespace, and between Pods. The network policy is a Namespace property. Network Admins can define policy in NSX through labels that can then be applied to pods or namespaces. Here we will show how the Kubernetes Network Policy definition causes the firewall rules to be automatically generated in NSX. By default, pods are non-isolated; they accept traffic from any source. Pods become isolated by having a NetworkPolicy that selects them. Once there is any NetworkPolicy in a namespace selecting a particular pod, that pod will reject any connections that are not allowed by any NetworkPolicy. Other pods in the namespace that are not selected by any NetworkPolicy will continue to accept all traffic. In our case, we will add a policy to only allow access to our nginx app from pods in a namespace with label app:db.
Create Network Policy

We will first check that there are no Network Policies created for this Namespace

1. Type `kubectl get NetworkPolicy`

Next we look at the network policy we want to create. This one establishes a rule about connectivity to pods with label app:nginx from namespaces with label app:db. Pods that are not in a namespace that matches the label will not be able to connect.

2. Type `cat nsx-demo-policy.yaml`

Let's apply that network policy

3. Type `kubectl apply -f nsx-demo-policy.yaml`

Let's see what we created

4. Type `kubectl get NetworkPolicy`
View Firewall Rules Created Automatically
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
From NSX-Mgr we can see that rules have been created based on our policy. NSX has dynamically created Source and Destination security groups and will apply the right policy

1. click on Firewall
2. Note the Network Policy Name and the scope being the Namespace we created it from.

**Traceflow**

NSX provides the capability to do detailed packet tracing across VMs and between pods. You can tell where a packet might have been dropped between two pods that you have deployed. We will deploy two pods in our namespace. We did not add any labels to our namespace when we created it, so our network policy should prevent communication between the two. Let's create the pods.

```
1. Type `kubectl apply -f /home/ubuntu/apps/nginx-sec.yaml`
```

**Configure Traceflow Source**

Return to NSX-Mgr in the Browser
1. Click on **Tools**
2. Select **Traceflow**
3. Choose the Logical Port and find a port with "db" in the name as the source
Configure Traceflow Destination

1. Under Destination, choose **Logical Port**
2. Choose one of the Ports with **Nginx** in the name
3. Click **Trace**

Verify Packets Are Dropped

1. The packet was dropped by the firewall.

Let's remove the network policy and try this again.
Remove Network Policy

```bash
ubuntu@ubuntu:~/apps$ kubectl delete -f nsx-demo-policy.yaml
networkpolicy "nsx-demo-policy" deleted
ubuntu@ubuntu:~/apps$ kubectl get networkpolicy
No resources found.
```

Return to the cli-vm

1. Type `kubectl delete -f nsx-demo-policy.yaml`
2. Type `kubectl get networkpolicy`

Re-Trace Your Application

1. Click the Re-Trace button
2. Once the network policy was removed, the packet made it to its destination successfully.

Traceflow is a very powerful capability that can also trace from VM to pod, VM and VM, and IP to IP. Try out a few more traces on your own.

Cleanup Deployments

```bash
ubuntu@ubuntu:~/apps$ kubectl delete -f /home/ubuntu/apps/nginx-sec.yaml
deployment "nginx" deleted
deployment "db" deleted
```
1. Type `kubectl delete -f /home/ubuntu/apps/nginx-sec.yaml`

1. Type `kubectl config set-context my-cluster --namespace default`

Returns the kubectl context to the default namespace.
Harbor Enterprise Container Registry

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. Harbor is included as an enterprise supported product with Pivotal Container Service (PKS). In this section, you will become familiar with the core capability of Harbor. You will create a project and see how to push and pull images from the repos. You will also enable content trust so that images are signed by the publisher and only signed images may be pulled from the project repo. You will also be introduced to the vulnerability scanning capability of Harbor. Most organizations will use a private registry rather than public Docker hub to improve security and latency for their applications. Although Harbor can be deployed as a highly available application, we have not done that for this lab.
Login to Harbor UI

1. Click on Google Chrome
2. Click on Harbor-01a.corp.local bookmark
3. Login to Harbor with Username: admin and Password: VMware1!
View Projects and Repositories

Harbor organizes images into a set of projects and repositories within those projects. Repositories can have one or more images associated with them. Each of the images are tagged. 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.

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 five different repos. The restreview repos will be used in Module 4 to deploy our restaurant review application.
1. Click on the [library/restreview-ui] repo

**View Image Vulnerability Summary**

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 4. Vulnerability scanning is part of PKS deployed Harbor registry.
1. Click on either of the images to see its vulnerability threat report.

**View Image Vulnerability Report**

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Severity</th>
<th>Package</th>
<th>Current version</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2017-1000408</td>
<td>unknown</td>
<td>glibc</td>
<td>2.19-18+deb8u6</td>
</tr>
<tr>
<td>CVE-2010-4756</td>
<td>negligible</td>
<td>glibc</td>
<td>2.19-18+deb8u6</td>
</tr>
<tr>
<td>CVE-2017-12133</td>
<td>medium</td>
<td>glibc</td>
<td>2.19-18+deb8u6</td>
</tr>
</tbody>
</table>

Each vulnerability is details, along with the package containing it, and the correct package version to fix the vulnerability.
Create Trusted Project

So far you have been using unsigned images. Now we want to have a production project that only contains images that are trusted. In order to do that we must sign the images. Let's start by creating a new project.

1. Click on Projects

Create New Project

1. Click on + Projects
1. Enter **trusted** for the project name and click **OK**

**Verify Project Created**

Note: The name of the project MUST be "trusted", in all lower case. We have tagged images with that path for you to use later in the lab. Using a different name will cause the image push to fail.

1. click on **trusted** to open your new project
Enable Content Trust on Your Project

1. Click on Configuration
We have options to Enable Content Trust and prevent vulnerable images from running. The image vulnerability restricts the pulling of images with CVEs that were identified in the image scans we saw previously. Enabling content trust means that only signed images can be pulled from this project.

1. **Enable content trust** and click **Save**

**Push Unsigned Image**

```
ubuntu@ubuntu:/apps5$ docker push harbor.corp.local/trusted/hello-world:v2
The push refers to repository [harbor.corp.local/trusted/hello-world]
5f70bf18e066: Pushed
76b0f8972821: Pushed
V2: digest: sha256:180ccaa4f259d3cd69431b1a3634b046ace131f46e323d1f7d907d399c62b size: 940
ubuntu@ubuntu:/apps5$ ll
```
1. Type `docker push harbor.corp.local/trusted/helloworld:V2`

We have an existing unsigned image that we want to push into our trusted project. Let's go back to the Harbor UI and see our image.

### View Unsigned Image

1. Click on **Repositories**
2. Click on the **arrow next to the Repo name** to see the individual image tags
3. Note that the image is unsigned

Now let's go back to the CLI

### Enable Docker Content Trust

```bash
ubuntu@ubuntu:~/apps$ export DOCKER_CONTENT_TRUST_SERVER=https://harbor.corp.local:4443
ubuntu@ubuntu:~/apps$ export DOCKER_CONTENT_TRUST=1
```

1. Type `export DOCKER_CONTENT_TRUST_SERVER=https://harbor.corp.local:4443`
2. Type `export DOCKER_CONTENT_TRUST=1`

These two commands enable image signing through Docker content trust and point to the Notary server. Our notary server is our Harbor registry.
Push Signed Image

1. Type `docker push harbor.corp.local/trusted/nginx:V2`
2. Type passphrase at all prompts: `handsonlab`

The root passphrase is only entered the first time you push a new image to your project. Note that you should not use the standard hol password 'VMware1!'. Docker notary doesn't seem to like the !. handsonlab was used as the password in testing.

Let's return to Harbor and see our signed image.

View Signed Image
1. Click on Refresh Icon in Harbor so your nginx image is visible.

1. Click on trusted/nginx image and verify that it was signed.

You may need to refresh the browser page to see your image. Let's create Kubernetes pods from our two images and see what happens. Return to the CLI.

**Create Pod From Unsigned Image**

```
ubuntu@ubuntu:~/apps$ kubectl apply -f /home/ubuntu/apps/hello-trusted-unsigned.yaml
replicationcontroller "helloworld-v2" created
```

```
NAME        READY STATUS      RESTARTS AGE
busy-5dd4648d9c-kbd6r  1/1   Running      0       36m
helloworld-v2-5whrg  0/1   ImagePullBackOff 0       5s
```

1. Type `kubectl apply -f /home/ubuntu/apps/hello-trusted-unsigned.yaml`
2. Type `kubectl get pods`

Notice that there was an error pulling the image. Let's investigate further.
Describe Pod To Find Error

1. Enter `kubectl describe po/helloworld-v2-#########`

Replace the ###### with the pod id from your previous kubectl get pods command. You can see why the pod failed to create. The image was not signed. Now let's run a pod with our signed image.

First let's clean up.

Clean Up Pod

1. Type `kubectl delete -f /home/ubuntu/apps/hello-trusted-unsigned.yaml`

This command will delete our deployment.

Create Pod From Signed Image

The first thing we need to do is create a secret. This will be mounted on our pod and shared with Harbor for authentication when pulling our image from the registry.

1. Type `kubectl create secret docker-registry regsecret`

   ```bash
   --docker-server=http://harbor.corp.local --docker-username=admin --docker-password=VMware1! --docker-email=admin@corp.local
   ```
The secret contains the information needed to login to the registry. Let's now take a look at the yaml file for our signed image.

**View Yaml To Create Pod From Signed Image**

```
ubuntu@ubuntu:~/apps$ cat nginx-trusted-signed.yaml
apiVersion: v1
kind: Service
metadata:  
  labels:  
    name: nginx
    name: nginx
spec:  
  ports:  
    - port: 80
  selector:  
    app: nginx
    type: LoadBalancer
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:  
  name: nginx
spec:  
  replicas: 1
  template:  
    metadata:  
      labels:  
        app: nginx
      spec:  
        containers:  
          - name: nginx
            image: harbor.com.local/trusted/nginx:v2
            ports:  
              - containerPort: 80
            imagePullSecrets:  
              - name: regsecret
```

1. Type `cat nginx-trusted-signed.yaml`

Note the imagePullSecrets refers to the secret we just created. Now we will create our pod from the signed image.
Create Pod

1. Type `kubectl apply -f nginx-trusted-signed.yaml`
2. Type `kubectl get pods`

Describe Pod To Verify Successful Image Pull

1. Type `kubectl describe po/nginx-#####` where ###### is the number for your pod in the get pods command
Clean Up Deployment

1. Type `kubectl delete -f nginx-trusted-signed.yaml`

   ubuntu@ubuntu:~/.apps$ kubectl delete -f nginx-trusted-signed.yaml
   service "nginx" deleted
   deployment "nginx" deleted
Conclusion

You should now have an understanding of how to operationalize Kubernetes using Pivotal Container Service (PKS)

You've finished Module 2

Congratulations on completing Module 2.

Proceed to any module below which interests you most.

- **Module 1 - Introduction to Kubernetes** (45 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 - 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 4 - 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 restaurant 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.
Your Lab Kubernetes Cluster

The command line tool used to interact with Kubernetes clusters is kubectl. If you took module 2 of the lab, you have some familiarity with using kubectl. We will dive deeper here. 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

![Connect to vCenter](image-url)
1. Click on **Google Chrome**
2. Select "**Use Windows Credentials**" and **Login**

(remember that corp\administrator is the vCenter user and VMware1! is the standard password in the Hands-on Labs if you ever need to enter one)

### Verify that All VMs are Running

You will notice that there are 6 VMs in the RegionA01-COMP01 cluster. These are the Master, two Worker nodes for your Kubernetes cluster, as well as the Harbor VM, the PKS Controller and Bosh. Harbor is VMware's container registry and is storing all of the container images used in this lab. More on that later. Bosh was started as part of the labstartup script. As part of PKS, Bosh monitors and maintains the health of the other VMs. So if one were to fail, it would be recreated by Bosh.

1. Note the **cli-vm** which will run all of our CLIs.
Connect to Kubernetes Cluster

You are now going to ssh into the cli-vm VM using Putty. For the purpose of this lab we have created a single client VM that will run the CLI's for PKS, kubectl and Bosh.

1. Click on **Putty** from your Windows Desktop
2. Select **cli-vm**
3. Click on **Load**
4. Click **Open**
Set kubectl Context Using PKS Controller

Execute the `pks login` command to authenticate to the pks API. You will use PKS to pull credential information from the deployed Kubernetes cluster and set the context for kubectl. The context associates cluster and authentication information to a context name and makes that name the current context. That's long way of saying that it points kubectl CLI to your cluster.

2. Type `pks get-credentials my-cluster`

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

2. Type `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. Type `kubectl get pods --namespace=kube-system`

Kubernetes can runs its system services as pods. With PKS deployed clusters, the master components run as processes managed by Bosh. Some of the supporting services run as pods. Let's take a look at those pods. Heapster aggregates cluster-wide monitoring and event data. The data is then pushed to influxdb for backend storage. Kubernetes also provides its own internal DNS server. This is used to provide domain names for communication between Kubernetes services. The Dashboard is the Kubernetes Management UI.

4. Type `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 and is included with PKS.

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.

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 created the context in the last section using pks get-credentials command. That command updated the file /home/ubuntu/.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

```
ubuntu@ubuntu-~$ cat /home/ubuntu/.kube/config

apiVersion: v1
clusters:
  - cluster:
      certificate-authority-data: LS0tLS1LCURudVI1BDRVJUSUZLQ0FURSU0LS0tCk1USURKRENQVQd20fSUZB01VSGVzaCgQz0FRBOwNz2VCCkYcgmVgvydHcwoGdwn2amxMzOJAGHeFb3q3gQ0tNWH2aUKYQlRrtekhhDrueUUEcOzIm9eE1plx3pORnQxckTisG0gKUVHhY21hLSUNhNz1os3J15RTVQO1a8N11tZBE1v5sBzRFG8tVhyUUmmt0YML3cyTH1URUWIZnN0GySbH5L4R2kNeHpkVFhdODlVY3U4ERKdK118V5bXNhL2pOMcJyWmN0qjyb090YlU2ymc60F8RzY6iSFHzvKSWBSBUqyVqFzFzBpL0jVCLz6p5CTXhna3Foa1lH0cucKFRc061Q8bMvzEJ9Xi6wKvpaVl36bNhCtNcOGzhFkVR2pamngTNG128cHLVv/9lOSvVNH1A6eFH2XtI45jFHLXZi0GEMARKNFfBTT17VcCjLRK
        server: https://10.40.14.54:8443
    name: my-cluster
  contexts:
    - context:
        name: my-cluster
        user: 22338d1a-07ed-4d03-a994-dd3f33d1c171
      kind: Config
      preferences: {}
    user:
      name: 22338d1a-07ed-4d03-a994-dd3f33d1c171
      user:
        token: eyVhBGo101JSU111Mc1m850C1eI5pXVCG9J.eCyvepSM101JpGWJlcm5ldGVE51Nhcn2pI2VY20NV850i1w3AEVzJX2znRyFy712i8nQ0V9yVvmi1j28HN2vVvDx50Lm5bWU5C1ytPrH5R0owQ0s05V2kL5bkHyKhntcN12ENmzVKNWNXvWekL21LCjrdWJlcm5ldGVEV5Knr1LUDWd3J1-WFsu-1h-CQHhBq0Bbta07090aDFHon-AKXcppU0J5NFFSHYl6HMKs2ver155GcH16qZnc1hN1jxkzhe6sh8XR6RqH
        HkxiiS2Zaffr2zSaSUR_t5q
```

The set-context command creates or appends a config file that is used by kubectl to interact with the cluster. 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 default namespace. View the contents of the config file.
1. Type `cat /home/ubuntu/.kube/config`

**Verify Config With kubectl**

```bash
ubuntu@ubuntu:~$ kubectl config view

apiVersion: v1
clusters:
  - cluster:
      certificate-authority-data: REDACTED
      name: my-cluster

default:
  cluster: my-cluster
  user: 22339d1a-07ed-4d03-a994-dd3f33d1c171

contexts:
  - context:
      cluster: my-cluster
      user: 22339d1a-07ed-4d03-a994-dd3f33d1c171
      name: my-cluster

current-context: my-cluster
kind: Config
preferences: {}
users:
  - name: 22339d1a-07ed-4d03-a994-dd3f33d1c171
    user:
      token: eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJpYjciOiJzdWIiLCJraWQiOiJtb2luZG93IiwibmFtZSI6InVzZXIiLCJpYXQiOjE1Mjg2NjYxNjUsImV4cCI6MTU4NDIyNjUxNn0.7N1E90HdThX2yZ9C0Hk8z4yZ2EYK89vnqVc2Vzdm1j2SiTy2NhVv5OZLbWbWU5CiIyMmjzO1Q5S0wN2TkLTRkMDMtcYTk5NC1L2VxKeiGUNDjJ9riWM3u-1h-Cq4Hmk0BtaAToT00DjTFh0n-XkkcpuU9JrNP95v2_6NH52verIHk112TZXa2fP22aSUR_t5Q
```

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

1. Type `kubectl config view`

**Namespaces**

```bash
ubuntu@ubuntu:~$ kubectl get namespaces

NAME              STATUS  AGE
 default           Active  14h
 kube-public       Active  14h
 kube-system       Active  14h
 pks-infrastructure Active 14h
```

Let's take a look at the namespaces in our cluster. What we care about for this lab are the kube-system and default namespaces. As we have previously seen, kube-system contains the Kubernetes cluster system objects. default will be where we are deploying our applications. If you took module 2 you probably also have the yelb-app namespace that was created as part of the networking section.
1. Type `kubectl get namespaces` 

```
ubuntu@ubuntu:~$ kubectl get pods
No resources found.
```

Now we will see how the namespaces label selector changes the output of the `get` commands. Our current context is default, and you have not created any application pods yet. So no resources are found. As you saw previously, the kube-system namespace is running the Kubernetes cluster system services.

1. Type `kubectl get pods` 
2. Type `kubectl get pods --namespace=kube-system` 

```
NAME                       READY STATUS      RESTARTS AGE
heapster-5cd77b4fcb-m99qv   1/1   Running     0          14h
kube-dns-7468555786-nn44k   3/3   Running     0          14h
kubernetes-dashboard-65f6f6c6c95-r28sf 1/1   Running     0          14h
monitoring-influxdb-679d8c75c-bdd5b 1/1   Running     0          14h
ubuntu@ubuntu:~$ 
```
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.
View Yaml Details

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.

1. Type `cd`
   
   This command will ensure that you are in the /home/ubuntu directory

2. Type `cat nginx.yml`
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. Version of the API
2. The deployment name is nginx.
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, 3 pods will be started. (Note: the scheduler will attempt to place them on separate nodes for availability but its best effort)
4. Notice that it has a Label, app: 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.
5. This pod is made up of a single container that will be instantiated based on the nginx:V1 image stored in the harbor.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.
yaml Service Spec

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

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

1. Type: LoadBalancer  By specifying LoadBalancer, NSX will create a Logical Load Balancer and associate an External IP to provide access to the service. 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: app:nginx  This is the label that the service uses to find the pods that it routes to.

Deploy nginx Application

```
ubuntu@ubuntu:~$ cd
ubuntu@ubuntu:~$ kubectl apply -f nginx.yml
service "nginx" created
deployment "nginx" created
ubuntu@ubuntu:~$ kubectl get deployment
NAME   DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
nginx  3        3        3            3          18s

ubuntu@ubuntu:~$ kubectl get pods
NAME                                               READY   STATUS   RESTARTS   AGE
nginx-57684cdcb6-bnvpr                             1/1      Running   0           23s
nginx-57684cdcb6-pvsk5                             1/1      Running   0           25s
nginx-57684cdcb6-qlkqlq                            1/1      Running   0           25s
```

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/ubuntu directory then type

1. Type `cd`
2. Type `kubectl create -f nginx.yml`
3. Type `kubectl get deployment`

Notice that the nginx deployment has a desired state of three pods and the current state is three running pods

4. Type `kubectl get pods`

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

**View the Service for 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

```
ubuntu@ubuntu:~$ kubectl get svc
NAME       TYPE          CLUSTER-IP     EXTERNAL-IP     PORT(S)        AGE
kubernetes  ClusterIP   10.100.200.1  <none>          443/TCP         15h
nginx       LoadBalancer 10.100.200.242 10.40.14.40      80:31737/TCP    2m
```

1. Type `kubectl get svc`
2. Look at the IP address for nginx. You will need it for the next step.

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 unless you are another service internal to the cluster. NSX has created a load balancer and allocated an external IP (10.40.14.40) that allows you to access the service and be routed to your service endpoints (pods). Your external IP may be different.
Access nginx Web Server

1. Click on Google Chrome
2. Enter http://10.40.14.40 or whatever external IP you saw in the previous kubectl get svc command for nginx

If you see the "Welcome to nginx" Your Web Server is running.
If you closed your CLI then

1. Click on Putty
2. Select cli-vm
3. Click on Load
4. Click Open

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. Type `kubectl get rs -o wide`  
2. Type `kubectl get pods -l app=nginx`

Notice the selector is based on the app=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. Type `kubectl scale deployment nginx --replicas 5`
2. Type `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 three copies of the nginx web server running to five replicas. The service automatically knows about the new endpoints and nsx-kube-proxy has updated 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.

<table>
<thead>
<tr>
<th>Type</th>
<th>kubectl scale deployment nginx --replicas 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>kubectl get pods</td>
</tr>
</tbody>
</table>

Delete Our Application

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

<table>
<thead>
<tr>
<th>Type</th>
<th>kubectl delete -f nginx.yml</th>
</tr>
</thead>
</table>

1. Type | kubectl delete -f nginx.yml |
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 4.

You've finished Module 3

Congratulations on completing Module 3

Proceed to any module below which interests you most.

- **Module 1 - Introduction to Kubernetes** (45 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 - Introduction to Pivotal Container Service (PKS)** (45 minutes) (Advanced) Operationalizing Kubernetes at scale is not for the faint of heart. PKS is a purpose built service to deploy and manage Kubernetes clusters. This module will introduce PKS, showing the capabilities to extend High Availability, scale, health monitoring and lifecycle management to Kubernetes.
• **Module 4 - 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 restaurant shop 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 4 - Deploy and Manage a Multi-Tiered Application (30 minutes)
Introduction

In this module you are going to deploy an application called yelb. It provides a simple capability to vote on your favorite restaurant. There is a front-end component called restreview-ui that fulfills a couple of roles. The first role is to host the Angular 2 application (i.e. the UI). When the browser connects to this layer it downloads the JavaScript code that builds the UI itself. Subsequent calls to other application components are proxied via the nginx service running the UI.

The rest-review appserver is a Sinatra application that reads and writes to a cache server (redis-server) as well as a Postgres backend database (restreview-db). Redis is used to store the number of page views whereas Postgres is used to persist the votes. As part of lab setup, container images have been built for you. If you are interested in the details, take lab HOL-1830-01 to dive into Docker in more depth.
This diagram represents the application we are going to manage. The application consists of four separate Kubernetes deployments, each with its own Load Balancer service. The frontend Web Server and a Redis Key Value store. The Redis store is implemented as a single Master with multiple workers. Each deployment defines a replica set for the underlying pods.
Deploy and Upgrade Restaurant Review Application to Add Persistent Volumes

We will deploy our restaurant review application and submit a few votes to see how it works. Our application is completely ephemeral. If a pod dies, all of its state is lost. Not what we want for an application that includes a database and cache. We will upgrade the application to take advantage of persistent volumes and verify that killing the pods does not remove the data.

Login To CLI
1. Click on Putty Icon
2. Select cli-vm
3. Click on Load
4. Click Open
View the Yaml Files

```
$ cd /home/ubuntu/apps
$ cat rest-review.yaml

# Node: this should work on an out of the box minikube setup
apiVersion: v1
kind: Service
metadata:
  name: redis-server
  labels:
    app: redis-server
    tier: cache
spec:
  type: ClusterIP
  ports:
    - port: 6379
  selector:
    app: redis-server
    tier: cache

---
apiVersion: v1
kind: Service
metadata:
  name: yelb-db
  labels:
    app: yelb-db
    tier: backend-db
spec:
  type: ClusterIP
  ports:
    - port: 5432
  selector:
    app: yelb-db
    tier: backend-db

---
apiVersion: v1
kind: Service
metadata:
  name: yelb-appserver
  labels:
    app: yelb-appserver
    tier: middletier
spec:
  type: ClusterIP
  ports:
    - port: 4567
  selector:
    app: yelb-appserver
    tier: middletier

---
apiVersion: v1
kind: Service
metadata:
  name: yelb-ui
```
In Module 3 we went through the details of the deployment, pod and service specs so we won't do that again here.

1. Type `cd /home/ubuntu/apps`
2. Type `cat rest-review.yaml`

Note that we can combine all of our deployments and services into a single file, and also notice that the image is `harbor.corp.local/library/restreview-ui:V1`.

**Deploy Restaurant Review V1 application**

Now you can deploy your application. This is done using the `kubectl apply` command and pointing to the appropriate yaml configuration files. You may have to run `get pods` a couple of times until the STATUS changes to running. Note: if you jumped straight to this module without doing any of the earlier modules, your `kubectl` context has not been set.
1. Execute the command `pkgs get-credentials my-cluster` to set the context.

2. Type `kubectl apply -f rest-review.yaml`  
This command creates all deployments and services defined in the Yaml. It will take a minute or so to come up. Verify its running by executing:

3. Type `kubectl get pods`  
View your deployment

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

5. Type `kubectl get rs`  
Describe the UI Pod For More Details

For details on your pod, you can describe it

1. Type `kubectl describe pods yelb-ui`  
The describe command is your first stop for troubleshooting a deployment. The event log at the bottom will often show you exactly what went wrong.
**Find External LoadBalancer IP .**

```
NAME     TYPE         CLUSTER-IP    EXTERNAL-IP    PORT(S)   AGE
kubernetes ClusterIP   10.100.200.1  <none>        443/TCP   19h
redis-server ClusterIP  10.100.200.243 <none>       6379/TCP   10m
yelp-appserver ClusterIP 10.100.200.204 <none>     8080/TCP   10m
yelp-db     ClusterIP   10.100.200.75  <none>       5432/TCP   10m
yelp-ui     LoadBalancer 10.100.200.18  10.40.14.40  80:32525/TCP 10m
```

Access the restaurant review application from your browser. The first step is to look at the service that is configured with the Load Balancer. In our case that is the UI service

1. Type `kubectl get svc`
2. Note the `EXTERNAL-IP`. That is the IP to get to the load balancer. Note that the load balancer port is 80

Return to the web browser to see the running application.

**View The Application**

1. Click on Google Chrome
2. Enter the **EXTERNAL-IP** from the `kubectl get svc`. It should be something like `10.40.14.4x`

You may get the nginx welcome page from the earlier module because we are using the same Load Balancer IP. Just refresh the page.

**Enter Votes in the Application**

The restaurant review application lets you vote as many times as you want for each restaurant. Try opening multiple browsers and voting from each of them. You will see that the application is caching the page views and persisting the vote totals to the Postgres database.
1. Click on as many votes as you would like.
2. Open Second browser tab, go to the application and try voting there as well.
Note the page views are increasing as well.

**Upgrade Application To Add Persistent Volumes**

Our application is completely ephemeral. If we delete the pods, all of the voting and
page view data is lost. We are going to a persistent volume, backed by a vSphere
Virtual Disk that has a lifecycle independent of the pods and VMs they are attached to.
For more information, check the storage section in Module 2 of this lab. We will see how
quickly and easily you are able to define the volume mount and 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.
We are going to make two changes to this application. The first is very simple. We will add "Version 2" text to the UI page. This was done by modifying the container image associated with the yelb-ui deployment. The second change is to add Volume mount information to the Redis-Server deployment yaml file. We will also add a Storage Policy and a Persistent Volume Claim that will be used by our Pods. When the new pods are created, their filesystem will be mounted on a persistent VMDK that was dynamically created. Note: our application stores only the Page Views in the Redis cache, the Voting information is in the Postgres container. We are upgrading the Redis cache container. It has no persistent volume until we upgrade, so our page view information is lost. The voting data stays because we are not upgrading that container and it continues to run.

```
ubuntu@ubuntu:~/apps$ cat rest-review-v2.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: yelb-ui
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: yelb-ui
        tier: frontend
    spec:
      containers:
        - name: yelb-ui
          image: harbor.corp.local/library/restreview-ui:v2
          ports:
            - containerPort: 80

---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: redis-server
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: redis-server
        tier: cache
    spec:
      containers:
        - name: redis-server
          image: harbor.corp.local/library/restreview-redis:v1
          ports:
            - containerPort: 6379
          volumeMounts:
            - name: redis-slave-data
              mountPath: /data
          volumes:
            - name: redis-slave-data
              persistentVolumeClaim:
                claimName: redis-slave-claim
```
1. Type `cat rest-review-v2.yaml`
2. Notice that the image changed to `harbor.corp.local/library/rest-review:v2`
3. Also notice that the redis-server spec includes a persistentVolumeClaim and where to mount the volume in the container.

**Storage Class And Persistent Volume Claim**

If you did not create the Storage Class and Persistent Volume Claim in Module 2, execute the following 3 commands:

1. Type `cd /home/ubuntu/apps`
2. Type `kubectl apply -f redis-sc.yaml`
3. Type `kubectl apply -f redis-slave-claim.yaml`

For more information on Kubernetes Storage Classes and Persistent Volume Claims, go to Module 2.

**Upgrade The Rest-review Deployment**

1. Type `kubectl apply --record=true -f rest-review-v2.yaml`

When we apply the new desired state to an existing deployment by changing its definition (in this case changing the container image that the pod is created with), Kubernetes will kill an old pod and add a new one. If we had multiple replicas running,
the application would continue to function because at least one pod would always be running.

2. Type `kubectl get pods`

You should see new pods creating and old terminating, but it happens fast. Try `kubectl get pods` until all are in STATUS Running.

**View Upgraded Application**

1. Click on your Chrome Browser
2. Refresh the Page and notice that the image is V2 and that your Votes and Page views are still there.
You may need to hold the shift key down while reloading the page to get the new page.

Now let's Delete the Redis server and database pods. The replication controller will restart them, so let's see if our page views are still there.

**Delete Redis Server and Database Pods**

1. Type `kubectl get pods` to find the name of the Redis Server pod
2. Type `kubectl delete pod redis-server-######` where the ###### are the id from `get pods`

Deleting the Redis application server

3. Type `kubectl delete pod yelb-db-#####` where the ###### is the id from `get pods`

Deleting the postgres database server

4. Type `kubectl get pods`

Notice the pods are terminating and new pods are created. The persistent volume will be reattached to the appserver pod but not the postgres database pod.
1. Refresh the browser page
2. Note that the Page Views have not changed

Remember that the actual votes were stored in our backend Postgres database, which we did not back with a persistent volume. So that data is gone. The page views were stored in our Redis cache and were backed by our persistent volume. So they survive the removal of the pods.

**Roll Back Restaurant Review 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 is a simple reverse of the upgrade process.

```
ubuntu@ubuntu:/apps$ kubectl rollout history deployment/yelb-ui
 deployments "yelb-ui"
REVISION  CHANGE-CAUSE
1  <none>
2  kubectl apply --record=true --filename=rest-review-v2.yaml
```

```
ubuntu@ubuntu:/apps$ kubectl rollout undo deployment/yelb-ui --to-revision 1
deployment "yelb-ui" rolled back
```

```
ubuntu@ubuntu:/apps$ kubectl get pods
NAME                  AGE     READY   STATUS    RESTARTS   TYPE
redis-server-689649c849-2mcb7 44m      1/1    Running   0          Terminating
yelb-appserver-85bfb784bc-q897c 44m      1/1    Running   0          Terminating
yelb-db-b567d4587-e8kltw  44m      1/1    Running   0          Terminating
yelb-ui-6447cc0d8-hcd52  9m       1/1    Terminating 0          Terminating
yelb-ui-69d3d984f-826z2  12s      1/1    Running   0          Terminating
```

1. Type `kubectl rollout history deployment/yelb-ui`

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.

```
1. Type kubectl rollout undo deployment/yelb-ui --to-revision 1
2. Type kubectl get pods
```

You should see terminating pods and new pods creating.
Refresh Browser

Once they are all running, go back to chrome and refresh the Browser again.

1. You should see that the Version 2 has been removed. Your page views plus any new votes that you added after the pod deletion should still be there.
Conclusion

You have now deployed a multi-tier application using Kubernetes and have rolled out an upgrade to that application without any downtime. You also saw that you could easily roll back to a previous version, also without downtime. If you have taken all four Modules, this concludes the Kubernetes and Pivotal Container Service (PKS) Lab.

You've finished Module 4

Congratulations on completing Module 4.

Proceed to any module below which interests you most.

- **Module 1 - Introduction to Kubernetes** (45 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 - Introduction to Pivotal Container Service (PKS)** (45 minutes) (Advanced) Operationalizing Kubernetes at scale is not for the faint of heart. PKS is a purpose built service to deploy and manage Kubernetes clusters. This module will introduce PKS, showing the capabilities to extend High Availability, scale, health monitoring and lifecycle management to Kubernetes.
• **Module 3 - 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.

### 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-1832-01-CNA

Version: 20180801-203534