# Table of Contents

Lab Overview - HOL-1830-01-CNA - Photon OS and Container Basics - Getting Started... 2
  Lab Guidance ................................................................................................................. 3
Module 1: Introduction to Containers (15 minutes)......................................................... 9
  Introduction to Containers ............................................................................................. 10
  Application Delivery with Containers .......................................................................... 16
  The Docker Revolution ............................................................................................... 28
  Conclusion ................................................................................................................... 30
Module 2: A Quick Tour of Docker (15 minutes) ............................................................ 31
  A Quick Tour of Docker .................................................................................................. 32
  The Dockerfile and docker build .................................................................................. 38
  Conclusion ................................................................................................................... 40
Module 3: Deploying a Docker Application on VMware PhotonOS (15 minutes)....... 41
  Photon OS .................................................................................................................... 42
  Deploying with Docker Compose .................................................................................. 44
  Stateless vs. Stateful Components ............................................................................... 50
  Conclusion ................................................................................................................... 60
Lab Overview - HOL-1830-01-CNA - Photon OS and Container Basics - Getting Started
Lab Guidance

Note: It will likely take more than 45 minutes to complete this lab and you may not finish all of the modules during your time. The modules are independent of each other so you can start at the beginning of any module and proceed from there. You can use the Table of Contents to access any module of your choosing.

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

Welcome to HOL-1830-01! In this lab, we will introduce the basic concepts of containers in general and Docker in particular. We will also look at how to implement a simple application in Docker and how to use container networking and Docker volumes. All exercises in this lab are run on top of vSphere and using PhotonOS™ as Container host OS. We will also briefly cover the vSphere Docker Volume Service.

Lab Module List:

- **Module 1 - Introduction to Containers** (15 minutes) (Basic) - In this module, we will explain containers and how they enable 3rd Platform application architectures to be run efficiently in distributed environments. We will also delve into the basics of Docker as a container platform. This is a reading module only without any interactive labs.

- **Module 2 - A Quick Tour of Docker** (15 minutes) (Basic) In this module, we're getting hands on with Docker and explore how to create simple containers and how to leverage Docker container networking. Finally, we'll look at how you can build your own images using a Dockerfile.

- **Module 3 - Deploying a Docker Application on VMware** (15 minutes) (Advanced) Here we will look at a more advanced example where we use Docker to deploy a simple application with a web-tier and a clustered database tier. We'll explore how to set up container networking and how to deploy using docker compose. Finally, we'll have a look at how to achieve data persistence across containers and container hosts.

Lab Captains:

- **Module 1 - Pontus Rydin, Principal System Engineer, USA**
- **Module 2 - Pontus Rydin, Principal System Engineer, USA**
- **Module 3 - Pontus Rydin, Principal System Engineer, USA**

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

[http://docs.hol.vmware.com](http://docs.hol.vmware.com)
This lab may be available in other languages. To set your language preference and have a localized manual deployed with your lab, you may utilize this document to help guide you through the process:


Location of the Main Console

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

Alternate Methods of Keyboard Data Entry

During this module, you will input text into the Main Console. Besides directly typing it in, there are two very helpful methods of entering data which make it easier to enter complex data.
Click and Drag Lab Manual Content Into Console Active Window

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

Accessing the Online International Keyboard

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

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

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

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

Click on the @ key

1. Click on the "@ key".

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

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

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

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

This cosmetic issue has no effect on your lab.

Look at the lower right portion of the screen
Please check to see that your lab is 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 Containers (15 minutes)
Introduction to Containers

In this Chapter, we will explain containers and how they enable 3rd Platform application architectures to be run efficiently in distributed environments.

Brief History of Containers

While containers are certainly a very popular topic right now, containers themselves are not new. They have existed for many years. FreeBSD, Solaris Zones, LXC...there are many incarnations of containerization technology.

You may ask - then why is Docker so popular? For a few good reasons, but mainly because Docker offers a very convenient form factor for distributing and sharing code. Very complex applications can be run using a single command line and the containerization ensures that the application behaves the same regardless of the underlying OS, hardware and networking environment. This has made it very popular among developers and testers who need to quickly spin up complex environments.

Containers are not without challenges. While they work extremely well in development, organizations are still struggling with issues like isolation, security, network virtualization and monitoring of containers. As you will discover throughout this lab, VMware is fully embracing the idea of concept of containers and is providing solutions to many of the problems we just discussed.
What are Containers?

Containers are an OS-level virtualization method in which the kernel of an operating system allows for multiple isolated user-space instances, instead of just one. The primary benefits of using containers include limited overhead, increased flexibility and efficient use of storage; the container looks like a regular OS instance from the user's perspective. Changes to the image can be made very quickly and pushed to a repository to share with others for further development and utilization.
What is Docker?

Docker containers wrap up a piece of software in a complete filesystem that contains everything it needs to run: code, runtime, system tools, system libraries – anything you can install on a server. This guarantees that it will always run the same, regardless of the environment it is running in.

Containers running on a single machine all share the same operating system kernel so they start instantly and make more efficient use of RAM. Images are constructed from layered filesystems so they can share common files, making disk usage and image downloads much more efficient. Docker containers are based on open standards allowing containers to run on all major Linux distributions and Microsoft operating systems.

Containers include the application and all of its dependencies, but share the kernel with other containers. They run as an isolated process in userspace on the host operating system.

Docker is a natural fit for microservice-based architectures.
How do Containers and Virtual Machines Differ?

This sounds a lot like virtual machines, doesn't it? Aren't they just variations on the same theme?

Well, not really... A virtual machine is an emulation of a piece of hardware. When you run a virtual machine, you run an entire operating system including kernel and BIOS that behaves as if it interacted with physical hardware. A container, on the other hand, runs in an emulated operating system with multiple containers sharing a single kernel. Containers run on top of a host operating system, which, in turn can run on either physical or virtual hardware.

A container is intended to run a single application. Containers are typically very specific, intended to run MySQL, Nginx, Redis, or some other application. So what happens if you need to run two distinct applications or services in a containerized environment? The recommendation is usually to use two separate containers. The low overhead and quick start-up times make running multiple containers trivial, thus they are typically scoped to a single application.

A virtual machine, on the other hand, has a broader range, and can run almost any operating system. As you are likely aware, the virtual machine serves as an extremely firm boundary between OS instances that's enforced by a robust hypervisor, and connects to Enterprise-level storage, network and compute systems in a trusted, well-defined and secure manner. Virtual machines have traditionally lent themselves to running 2nd Platform (Web - App - Database) applications that compromise 99% of the application space today.
Containers provide great application portability, enabling the consistent provisioning of the application across infrastructures. However, applications and data alone are rarely the major barrier to workload mobility. Instead, operational requirements such as performance and capacity management, security, and various management tool integrations can make redeploying workloads to new environments a significant challenge. So while containers help with portability, they’re again only a piece of a bigger puzzle.

Due to the fundamental differences in architecture (namely the ESXi hypervisor used by virtual machines versus the shared kernel space leveraged by containers), Linux containers will not achieve the same level of isolation and security. Furthermore, the toolsets available in the virtual machine ecosystem are battle-tested and Enterprise-grade, enabling scores of benefits (stability, compliance, integrated operations, etc) that are indispensable to operations and infrastructure teams.

For these reasons, VMware provides the best of both worlds by offering an optimized operating system built for containers to run with minimal overhead. By dedicating an
extremely lightweight operating system to run containerized workloads, we don't have to choose one or the other - we can have both! By taking advantage of memory sharing, a core feature of the ESXi hypervisor, we drastically reduce the operating system overhead while enabling the application flexibility promised by containers.

In lab HOL-1830-02 we will look at vSphere Integrated Containers, which provides enterprise level isolation, security and networking for containers by turning them into virtual machine-like objects that can be managed just like virtual machines.
Application Delivery with Containers

Notice: This section is all reading. If you are eager to get to typing on the keyboard, you can move on to the next module, but you will miss out on learning some very important concepts.

Containers enables you do deliver applications as something called micro services. This isn't a new concept. In fact, it's quite old. Back in the mainframe days if you used something like IBM CICS, your code would be divided into transactions, each responsible for carrying out some specific business function. For example, withdrawing money from an account would be one transaction, depositing would be another and so on. Similar ideas appeared in the 90s with CORBA and the early 2000s with SOA.

At its core, the concept is simple: Each business function is encapsulated in a self-contained service that exposes a well-defined and published interface describing how to interact with it. Since the services are self-contained and independent, they can be managed and upgraded independently. For example, if the credit card functionality of an application needs to be replaced, we can just replace the corresponding micro service. As long as we expose the same interface, any application component using the service will continue running without even noticing the change.

As we discussed above, this is nothing new. What's new is that the container form factor makes delivering and managing micro services a lot easier.

Application Development and Delivery
If we look at the Outcomes Delivered from a new model of IT, Businesses are increasing their focus on **App and Infrastructure Delivery Automation** throughout the datacenter.

**App and Infrastructure Delivery Automation**

- Faster delivery of apps and IT services with support for continuous updates
- Automated Infrastructure and application provisioning across multiple hardware platforms, hypervisors and clouds, Self service

**Key Impact Metric**
IT Service Delivery in Minutes

IT is making strides to provide the ability to enable faster delivery of application and IT Services leveraging capabilities derived from automated infrastructure and application provisioning.

**New Business Imperative**

**Create amazing experiences for customers**
- Big data to improve customer experience
- Systems of engagement to delight users
- Mobile applications to interact with customers wherever they are

**Deliver and update products continuously**
- Everything as a service to simplify customer adoption and upgrades
- Agile development to evolve applications quickly
- Open systems to embrace new technology

Competitive businesses are delivering new applications to market in increasingly faster cycles, ushering in technologies like Linux containers and microservices. Next-
generation applications are being built on infrastructure assumed to be dynamic and elastic. To keep our customers agile, our Cloud-Native Apps group builds infrastructure technologies to open, common standards that preserve security, performance, and ease-of-use, from developer desktop to the production stack.

**Moving Faster Requires Design and Culture Changes**

To move faster, businesses implement a variety of cultural, design, and engineering changes. At VMware, we are striving to make the Developer a first class citizen of the Data Center and help align them with IT’s journey to achieve streamlined App and Infrastructure Delivery Automation.

**History of Platforms**

To move faster, businesses implement a variety of cultural, design, and engineering changes. At VMware, we are striving to make the Developer a first class citizen of the Data Center and help align them with IT’s journey to achieve streamlined App and Infrastructure Delivery Automation.
1st Platform systems were based around mainframes and traditional servers without virtualization. Consolidation was a serious issue and it was normal to run one application per physical server.

2nd Platform architectures have been the standard mode for quite a while. This is the traditional Client/Server/Database model with which you are likely very familiar, leveraging the virtualization of x86 hardware to increase consolidation ratios, add high availability and extremely flexible and powerful management of workloads.

3rd Platform moves up the stack, standardizing on Linux Operating Systems primarily, which allows developers to focus on the application exclusively. Portability, scalability and highly dynamic environments are valued highly in this space. We will focus on this for the rest of the module.

3rd Platform - Microservice Architecture

Microservices are growing in popularity, due in no small part to companies like Netflix and Paypal that have embraced this relatively new model. When we consider microservices, we need to understand both the benefits and the limitations inherent in the model, as well as ensure we fully understand the business drivers.

At its heart, microservice architecture is about doing one thing and doing it well. Each microservice has one job. This is clearly in stark contrast to the monolithic applications many of us are used to; using microservices, we can update components of the application quickly without forcing a full recompile of the entire application. But it is not a "free ride" - this model poses new challenges to application developers and operations teams as many assumptions no longer hold true.
The recent rise of containerization has directly contributed to the uptake of microservices, as it is now very easy to quickly spin up a new, lightweight run-time environments for the application.

The ability to provide single-purpose components with clean APIs between them is an essential design requirement for microservices architecture. At their core, microservices have two main characteristics; they are stateless and distributed. To achieve this, let's take a closer look at the Twelve-Factor App methodology in more detail to help explain microservices architecture as a whole.

The Twelve-Factor App

To allow the developer maximum flexibility in their choice of programming languages and back-end services, Software-as-a-Service web applications should be designed with the following characteristics:

- Use of a declarative format to attempt to minimize or eliminate side effects by describing what the program should accomplish, rather than describing how to go about it. At a high level it’s the variance between a section of code and a configuration file.
- Clean Contract with the underlying Operating Systems which enables portability to run and execute on any infrastructure. API’s are commonly used to achieve this functionality.
- Ability to be deployed into modern cloud platforms; removing the dependencies on underlying hardware and platform.
- Keep development, staging, and production as similar as possible. Minimize the deviation between the two environments for continuous development.
- Ability to scale up (and down) as the application requires without needing to change the tool sets, architecture or development practices.

At a high level, the 12 Factors that are used to achieve these characteristics are:

1. **Codebase** - One codebase tracked in revision control, many deploys
2. **Dependencies** - Explicitly declare and isolate dependencies
3. **Config** - Store config in the environment
4. **Backing Services** - Treat backing services as attached resources
5. **Build, release, run** - Strictly separate build and run stages
6. **Process** - Execute the app as one or more stateless processes
7. **Port Binding** - Export services via port binding
8. **Concurrency** - Scale out via the process model
9. **Disposability** - Maximize robustness with fast startup and graceful shutdown
10. **Dev/Pro Parity** - Keep development, staging, and production as similar as possible
11. **Logs** - Treat logs as event streams
12. **Admin Process** - Run admin/management tasks as one-off processes

For additional detailed information on these factors, check out [12factor.net](http://12factor.net).

**Benefits of Microservices**

Microservice architecture has benefits and challenges. If the development and operating models in the company do not change, or only partially change, things could get muddled very quickly. Decomposing an existing app into hundreds of independent services requires some choreography and a well thought-out plan. So why are teams considering this move? Because there are considerable benefits!
Resilience

With a properly architected microservice-based application, the individual services will function similarly to a bulkhead in a ship. Individual components can fail, but this does not mean the ship will sink. The following tenet is held closely by many development teams - "Fail fast, fail often." The quicker a team is able to identify a malfunctioning module, the faster they can repair it and return to full operation.

Consider an online music player application - as a user, I might only care about playing artists in my library. The loss of the search functionality may not bother me at all. In the event that the Search service goes down, it would be nice if the rest of the application stays functional. The dev team is then able to fix the misbehaving feature independently of the rest of the application.

Defining "Service Boundaries" is important when architecting a microservice-based application!
Scaling

If a particular service is causing latency in your application, it's trivial to scale up instances of that specific service if the application is designed to take full advantage of microservices. This is a huge improvement over monolithic applications.

Similar to the Resilience topic, with a monolithic application, one poorly-performing component can slow down the entire application. With microservices, it is almost trivial to scale up the service that is causing the latency. Once again, this scalability must be built into the application's DNA to function properly.
Deployment

Once again, microservices allow components to be upgraded and even changed out for entirely new, heterogeneous pieces of technology without bringing down the entire application. Netflix pushes updates constantly to production code in exactly this manner.

Misbehaving code can be isolated and rolled back immediately. Upgrades can be pushed out, tested, and either rolled back or pushed out further if they have been successful.
"Organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations" --Melvin Conway

The underlying premise here is that the application should align to the business drivers, not to the fragmentation of the teams. Microservices allow for the creation of right-sized, more flexible teams that can more easily align to the business drivers behind the application. Hence, ideas like the "two pizza rule" in which teams should be limited to the number of people that can finish two pizzas in a sitting (conventional wisdom says this is eight or less...though my personal research has proved two pizzas do not feed more than four people.)
No Silver Bullet!

Microservices can be accompanied by additional operations overhead compared to the monolithic application provisioned to a application server cluster. When each service is separately built out, they could each potentially require clustering for fail over and high availability. When you add in load balancing, logging and messaging layers between these services, the real-estate starts to become sizable even in comparison to a large off the shelf application. Microservices also require a considerable amount of DevOps and Release Automation skills. The responsibility of ownership of the application does not end when the code is released into production, the Developer of the application essentially owns the application until it is retired. The natural evolution of the code and collaborative style in which it is developed can lend itself to challenges when making a major change to the components of the application. This can be partially solved with backwards compatibility but it is not the panacea that some in the industry may claim.

Microservices can only be utilized in certain use cases and even then, Microservices open up a world of new possibilities that come with new challenges and operational hurdles. How do we handle stateful services? What about orchestration? What is the best way to store data in this model? How do we guarantee a data persistence model? Precisely how do I scale an application properly? What about “simple” things like DNS and content management? Some of these questions do not have definitive solutions yet. A distributed system can also introduce a new level of complexity that may not have been such a large concern like network latency, fault tolerance, versioning, and unpredictable loads in the application. The operational cost of application developers needing to consider these potential issues in new scenarios can be high and should be expected throughout the development process.

When considering the adoption of a Microservices, ensure that the use case is sound, the team is aware of the potential challenges and above all, the benefits of this model outweigh the cost.
Recommended reading: If you would like to learn more about the operational and feasibility considerations of Microservices, look up Benjamin Wootton and read some of his publications on the topic, specifically 'Microservices - Not A Free Lunch!'.
The Docker Revolution

In this lesson we will discuss the tremendous impact containers are having on the IT industry and what the benefits and challenges.

What Just Happened?

The reason you're taking this lab is probably that you've heard talk of containers swirling around and are eager to understand what all the buzz is about. Let's take a minute to try to understand what has happened over the last few years.

Revenge of the Developer

Some of the most successful startups today are driven by developers and coders rather than business people. This is a radical change from how things used to be done. The business used to come up with ideas and push them down to development to have them translated into code. That is about to change and has already changed in some industries. Developers now constantly push the boundaries of what's possible and send their ideas to the business who builds offerings around them. This shift fundamentally changes how IT operates and has driven the demand for new tools and technologies.

From Months to Minutes

Under the old paradigm, releases once every quarter was considered frequent. Today, some industries need to make several releases an hour to keep up. Also, it would be impossible to roll out three months worth of changes to a system that's constantly used by millions of people and that doesn't tolerate any downtime. Frequent changes means smaller changes, which means easier rollouts. In addition, if you keep the changes well encapsulated and independent from each other, they become much easier to roll back if something should go wrong.

It's been reported that the big social media and web portal companies deploy hundreds of changes every day. Clearly, something in the way we build and deploy code has to change fundamentally!

DevOps and CI/CD
To get to the speed and agility needed, we need to change our processes fundamentally.

You may have heard of DevOps. This is the idea that we can no longer have developers simply throwing code over the fence to operations and hope that it gets deployed. It's too slow and error prone. Instead we need to streamline the handover and even eliminate it. DevOps is the idea of integrating development and operations so they behave as one unit.

Sounds good, but how do we get there? One way is to implement a Continuous Integration/Continuous Deployment process. The idea is that whenever a developer commits a piece of code and tags it as ready for release, the deployment process automatically kicks off. In some cases, this process can be made completely automatic by introducing automated testing and deployment.

The challenge to implementing a CI/CD pipeline is that it requires you to set up and tear down complete environments very quickly.

**Docker to the Rescue!**

Docker is no silver bullet, but it offers some attractive features helping anyone who is trying to increase speed and agility:

- Containers are fast to deploy and undeploy
- The container form factor is convenient for shipping small changes
- It’s relatively easy to use containers as part of a CI/CD pipeline
- Container schedulers such as Kubernetes facilitate low-impact upgrades
- Auto-scaling becomes easier, since containers can be created and destroyed very quickly

**The Challenges**

So why isn't everyone doing this right now? Because problems arise when you try to operationalize this model. There are currently no good tools for monitoring and managing containers. Also, most IT operations organizations lack the knowledge and processes for handling containers. Some of the challenges IT operations is struggling with are:

- Isolation between containers is weaker than between virtual machines.
- Backup and restore of containers is still immature.
- Network virtualization and security are still very immature.
- Kubernetes is great for orchestration and scheduling but difficult to manage.
- No good monitoring solution for containers exists.

VMware is aggressively working to mitigate these problems using products like vSphere Integrated Containers, Harbor and Admiral.
Conclusion

You've finished Module 1!

Congratulations on completing Module 1. You should now have a solid foundational understanding of container technologies and the Docker implementation.

Based on your interests, please proceed to any module below:

- Module 2: A Quick Tour of Docker (15 minutes/basic)
- Module 3: Deploying a Docker application on VMware PhotonOS (15 minutes/advanced)

How to End Lab

To end your lab click on the **END** button.
Module 2: A Quick Tour of Docker (15 minutes)
A Quick Tour of Docker

In this lesson we will cover the following:

- Starting a simple container
- Running a simple service as a container
- Container Networking

Starting our first container

Enough theory for now! Let's get a container up and running! To begin we're going to SSH from our Windows console to a VM running Photon OS - we'll cover Photon OS in detail a bit later in the lab, but know that Photon OS is an open-sourced, lightweight Linux distribution optimized for virtualized environments and containers. From there we will run our Docker commands.

Here are the steps:

1. Click on the Docker CLI icon on the desktop.
2. At the login prompt, type `root` followed by Enter.
3. Ifconfig eth0
4. `docker run -it busybox`

1. Click on the Docker CLI icon on the desktop.
2. At the login prompt, type `root` followed by Enter.
3. At the command line, type the following two commands, each followed by Enter:
   ```
   ls
   ```
4. `ifconfig eth0`
5. Take note of the list of files and the network address for eth0. It should be on a 192.168.x.x network.
6. At the command prompt, type the command below, followed by Enter:
   ```
   docker run -it busybox
   ```

## Explore the world inside the container

1. You are at a command prompt inside the container. Type the following two commands, each followed by Enter:
   ```
   ls
   ```
2. `ifconfig eth0`
   - Notice that you ended up in a different file system.
   - Notice that your primary NIC (eth0) is now on a 172.x.x.x network.
3. Type `exit` followed by Enter. You are now back to the command prompt of the host VM.

## What happened?

Congratulations! You just created and ran your first container! But what actually happened?

The command `docker run -it busybox` told docker to run a Busybox environment. This is a tiny Linux environment typically used for smaller devices, such as routers. If the docker image for busybox didn't exist in your local cache, it would have downloaded it. To make things quicker, we made sure you had it cached. The `-it` flag simply meant that you want an interactive process with the current terminal connected to it.

So this dropped you into the shell of busybox. You also noticed that you were given a different file system. This is because each Docker container comes with its own file
system that's completely isolated from the host. You also noticed you were connected to the 172.x.x.x network instead of the 192.168.x.x one. This is because docker creates an ad-hoc virtual network for your containers so that ports you open in different containers won't interfere with each other.

**Let's define some terms!**

We've already used the terms container and image. Let's define them and some other concepts a bit more in detail.

- **Image** - The code you are running in a container packaged as a virtual file system known as a "layered file system". More about that later.
- **Container** - The running instance of an image. This is somewhat analogous to a virtual machine but is isolated at the OS level rather than using virtualized hardware. Each container has its own file system and virtual network.
- **Repository** - An image or set of images packaged in such a way that they can be published and reused.
- **Registry** - A catalog service for repositories. When you type `docker run -it busybox`, a query is sent to a registry service to locate the image(s) making up busybox. The registry server delivers the content of the images if they're not already in the local cache.
- **Docker host** - The machine (virtual and physical) hosting your containers.

**Starting a service in a container**

```
docker run -d -p 80:80 nginx
```

![Image showing the process of starting a service in a container](image)

```
curl http://localhost
```

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.
Let's get hands on again and do something a bit more useful. How about starting an nginx web proxy? That can be a bit of a daunting task when doing it in a traditional VM. With Docker, it's a bit easier. Let's do it:

1. Type this command, followed by enter:
   
   ```bash
   docker run -d -p 80:80 nginx
   ```

2. You have now started an nginx web proxy! Let's see if it responds, Type this command, followed by enter:

   ```bash
   curl localhost
   ```

   You should see a short HTML document printed in your terminal window.

### Daemons and port mappings

You probably noticed two things: You weren't dropped into the command prompt of the container and you could access the service on the http port (80) on the Docker host itself (localhost). It's all in the command options!

- `-d` means detached mode. In other words, the container is started as a background process and doesn't take over the terminal window.
- `-p` causes ports to the private container network to be mapped to ports on the docker host.
- `-p 80:80` simply means "map port 80 on the private network to port 80 on the docker host".

Speaking of private container networks, let's have a quick look at how that works.
1. Type this command followed by Enter:
   ```bash
   ifconfig docker0
   ```

2. Docker has created a virtual network interface for the containers running on this host. Take note of the "inet addr" of this interface. It should be on the 172.x.x.x network.

3. Type this command followed by Enter:
   ```bash
   route
   ```

4. This will display the routing table of the current host. Notice that you will see one or more 172.x.x.x addresses in this table. These entries were added by Docker to inform the host how to route traffic to the private container network.

Clean up

Let's clean up after this exercise and get ready for the next!

1. Type the following command, followed by Enter:
   ```bash
   /root/cleanup.sh
   ```

Creating a container network

1. Type the following command, followed by Enter:
   ```bash
   docker network create hol-net
   ```

2. Type the following command, followed by Enter:
   ```bash
   docker run -d --network hol-net --name web nginx
   ```

3. Type the following command, followed by Enter:
   ```bash
   docker run -it --network hol-net busybox
   ```

4. Type the following command, followed by Enter:
   ```bash
   ping -c 1 web
   ```

5. Type the following command, followed by Enter:
   ```bash
   /root/cleanup.sh
   ```

6. Type the following command, followed by Enter:
   ```bash
   docker network rm hol-net
   ```
We learned in the previous section how Docker automatically creates a private container network for communication between your containers. But what if we want to create a separate network for a specific group of containers? That's where the `docker network create` command comes in handy!

Let's spin up a new network, then create an nginx proxy and a busybox and hook them both up to our newly created network.

1. Let's start with the container network. Type this command, followed by Enter:
   ```bash
docker network create hol-net
   ```
2. We now have a new container network called hol-net. Let's start an nginx proxy and connect it to that network! Type this command followed by Enter:
   ```bash
docker run -d --network hol-net --name web nginx
   ```
   The `--name` parameter assigns a name to the container. This name can be used as a host name for containers wanting to connect to the container. We're calling container "web".
3. Let's start the busybox! Type this command followed by Enter:
   ```bash
docker run -it --network hol-net busybox
   ```
4. You will now drop into the shell of the busybox. Let's see if we can communicate with the nginx proxy. Remember we gave it the name "web", so we should be able to use that. Type this command, followed by enter:
   ```bash
   ping -c 1 web
   ```
5. You should see a successful packet transmission. Now let's clean up. Type these three commands, each followed by Enter.
   ```bash
   exit
   /root/cleanup.sh
   docker network rm hol-net
   ```
The Dockerfile and docker build

So far, we've only been using pre-built images. But how were those images created in the first place? And what if you want to build your own customized image?

That's where the docker build command comes into the picture. This command reads a definition file called the Dockerfile and uses the instructions in the file to build a new image. You can see the Dockerfile as the source code of an image.

Building a "Hello world" image

1. If you're not already logged in, double-click on the Docker-CLI icon and log in as root.
2. Examine the Dockerfile by typing the following two commands and pressing Enter after each line.
   ```
   cd /root/hello
   cat Dockerfile
   ```
3. There are three directives in this Dockerfile:
   1. FROM - Tells docker which image to base this on. In our case, it's busybox.
   2. COPY - This copies the file hello.sh from our file system into the file system of the image.
   3. ENTRYPOINT - Tells docker which file to run when the container is started.
   4. docker build -t hello .
   5. docker run hello

Let's build a simple application inside an image. Our "code" is the file "hello.sh" and our job is to build a Container based on busybox, copying our application code to it and instruct Docker to run that code when the container is started.

1. If you're not already logged in, double-click on the Docker-CLI icon and log in as root.
2. Examine the Dockerfile by typing the following two commands and pressing Enter after each line.
   ```
   cd /root/hello
   cat Dockerfile
   ```
3. There are three directives in this Dockerfile:
   1. FROM - Tells docker which image to base this on. In our case, it's busybox.
   2. COPY - This copies the file hello.sh from our file system into the file system of the image.
   3. ENTRYPOINT - Tells docker which file to run when the container is started.
5. Let’s build the new image! Type the following and press Enter. (Notice the dot at the end!)

   `docker build -t hello .`

6. The -t option allows us to give the image a name (tag). We'll use that when running it.

7. Run the Hello world application. Type the following and press Enter:

   `docker run hello`

---

**Clean up**

1. Type the following command and press Enter:

   `/root/cleanup.sh`
Conclusion

In this module, we have covered the following:

- Understanding basic docker commands
- Building a docker image from a Dockerfile

You've finished Module 2

Congratulations on completing Module 2.

Proceed to any module below which interests you most. [Add any custom/optional information for your lab manual.]

- Module 3: Deploying a Docker Application on VMware PhotonOS (15 minutes)

How to End Lab

To end your lab click on the END button.
Module 3: Deploying a Docker Application on VMware PhotonOS (15 minutes)
Photon OS

An introduction to VMware Photon OS, a container host operating system optimized for vSphere.

Introduction to VMware Photon OS

All the exercises in this module are running on top of Photon OS. Let's take a couple of minutes to understand what Photon OS is and why VMware decided to offer a container host OS.

Photon OS is a lightweight Linux operating system for Cloud-Native apps. Photon OS is optimized for vSphere and vSphere-based cloud offerings providing an easy way for our customers to extend their current platform with VMware and run modern, distributed applications using containers.

Photon provides the following benefits:

• Support for the most popular Linux container formats including Docker, rkt, and Garden from Pivotal
• Minimal footprint (approximately 300MB), to provide an efficient environment for running containers
• Seamless migration of container workloads from development to production
• All the security, management, and orchestration benefits already provided with vSphere offering system administrators with operational simplicity

We have open sourced Photon OS to encourage widespread contributions and testing from customers, partners, prospects, and the developer community at large. It is available today on GitHub for forking and experimentation; the binary is also available on JFrog Bintray.

By offering Photon OS, we are able to provide integrated support for all aspects of the infrastructure, adding to the leading compute, storage, networking, and management found today. Customers will benefit from end-to-end testing, compatibility, and interoperability with the rest of our software-defined data center and End User Computing product portfolios. Through integration between Photon and Lightwave,
customers can enforce security and governance on container workloads, for example, by ensuring only authorized containers are run on authorized hosts by authorized users.
Deploying with Docker Compose

In this lesson, we will learn how to deploy a multi-node application using the docker-compose tool. Specifically, we will discuss:

- Structure of the application
- The docker-compose.yml file
- How to start the application
- Testing the application

Structure of the application

In this section, we will deploy a more meaningful application. We're going to build a simple product rating system for a storefront website. The customer feedback is stored in a replicated Redis datastore and accessed through a PHP-based front end. We are going to deploy each of these three components in its own container and make them all communicate on a private container network. The PHP frontend will expose port 80 to the public network.

One of the challenges we're going to face is how to deploy a multi-tier application without having to manually deploy each tier and carefully tying them together. Luckily,
there are several tools for doing that in an enterprise environment. Kubernetes is probably the most well-known one in that space.

In this exercise, we're going for a simpler and more lightweight solution using Docker Compose. This tool lets us define the container making up the tiers of the application in a single YAML file and bring the entire application up using a single command.

**Examining docker-compose.yml**

We have already provided a docker-compose file for you to use with this exercise. Let's familiarize ourselves with the structure of this file.

1. If you are not already logged into the Docker CLI, double-click the Docker-CLI icon on the desktop and log in as root (no password needed)
2. Type the following two commands, each followed by Enter:
   - `cd /root/vhobby`
   - `cat docker-compose.yml`
3. You should see a short text file. Let's examine the contents!

   • **services** - Lists the services (containers) the application will spin up.
   • **web** - This is our first service that we call "web" which will implement the web frontend.
   • **links** - Specifies our dependencies. We are dependent on both the master and the slave redis nodes.
   • **image** - The name of the docker image to spin up when the service starts.
   • **ports** - Any ports we want to expose publicly. In this case, we're mapping the internal port 80 to port 80 on the public network.
   • **redis-master, redis-slave** - Service definitions for the redis nodes.

Since we have dependencies between the nodes, Docker will automatically place them on a container network and link them together. It will also set up a simple naming
service so that the code in the frontend can refer to the nodes by name instead of using IP addresses.

**Starting the application**

![Docker-Compose Command]

Now for the fun (and easy) part. Let's start the application. Since the application is fully described by the docker-compose file, we can bring it up without adding any additional parameters.

1. Type the following command followed by Enter:
   ```bash
docker-compose up -d
   ```
2. You should see some status messages and the application should be ready for use within a few seconds.

The -d option just tells docker-compose to start the containers in background mode without assigning the current terminal to them.
Testing the application

Let's test the application!

1. From the start menu, select Chrome.
Navigating to the storefront

1. In the address field of Chrome, type the following and press enter: **192.168.100.104**
2. You should now see a product review page. Enter a review of a product and click Submit.

The application will write the product review to the master redis node and then read it back from the slave one. This demonstrates the behavior of a simple multi-tier application implemented as containers. Given the modular design of the application and the use of micro-services, we can replace individual components with no or little downtime.
Bringing the application down

Let's clean up after ourselves by bringing the application down!

1. Type the following and press Enter:
   ```
docker-compose down
   ```

Not surprisingly, the "down" command does the exact opposite of "up" and brings down the containers, as well as releasing all resources associated with them.
Stateless vs. Stateful Components

In this section we will explore stateful components and how to use volumes to store state.

Within the context of application services, the term "state" typically refers to data that needs to be preserved between function calls. For example, the contents of a shopping cart in a web store is a typical example of state. Some state is said to be persistent, which means that it must survive system restarts and crashes. Typically, that means that it has to be written to some secondary storage, such as a disk.

By default, when you run a container, it creates its own ad hoc file system that stores data as long as the container is active. If you shut down the container and delete it, the data is forever lost. This is probably not what you want when running e.g. a database inside a container. To solve this problem, you can specify volumes that are stored externally in the file system of the Docker host.

Starting a container with an external volume

Let's create a simple container with an external volume!

1. If not already logged in, double-click the Docker-CLI icon on the desktop and log in as root (no password needed)
2. Type the following and press Enter:
   ```bash
docker run -it -v /data/busybox:/data busybox
   ```
3. You have now told Docker to run the busybox image and map the directory `/data/busybox` on your Docker host to the directory `/data` inside the container. Let's create some data! Type the following and press Enter:
   ```bash
cd data
   echo "Kilroy was here!" > test.txt
   ```
4. Type the following and press Enter:
   ```bash
echo "Kilroy was here" > test.txt
   ```
5. Exit the container by typing the following and pressing Enter:
   ```bash
   exit
   ```

### Examining the volume

We are now back in the host. Let's look at the directory we mapped by typing the following and pressing Enter.

1. ```bash
   ls /data/busybox
   ```
   You will see that the file "test.txt" appeared in the directory. This is because it was mapped to /data in the container and we created a file there from inside the container.

2. Let's spin up a new container and map it to the same volume. Type the following and press Enter:
   ```bash
   docker run -it -v /data/busybox:/data busybox
   ```

3. Verify that we can see the file we created in the other container by typing the following and pressing Enter:
   ```bash
   cat /data/test.txt
   ```

4. You should now see the test message we created in the previous container. Now exit the container and clean up by typing the following and pressing Enter after each line:
   ```bash
   exit
   ```

5. ```bash
   /root/cleanup.sh
   ```
Using volumes with Docker Compose

Let's revisit our product rating application! Obviously, we'd like the product ratings we enter to survive container shutdowns or crashes. To do that, we need to assign external volumes to the two Redis nodes. Let's have a look at how that's done!

1. Type the following and press Enter after each line:
   ```bash
   cd /root/vhobby/vol
   cat docker-compose.yml
   ```
2. You should now see a short text file. Notice that the redis-master and the redis-slave both define their own external volumes. You can list multiple volume mappings if needed.

Running the demo app with persistent state
Let's try to run the demo app with persistent state backed by volumes! We have entered a couple of product reviews as we prepared the lab and since we're keeping persistent state in this version of the app, you should be able to see them.

1. Make sure you're in the right directory by typing the following and pressing Enter:
   ```
   cd /root/vhobby/vol
   ```
2. Start the application. Type the following and press Enter:
   ```
   docker-compose up -d
   ```

**Verifying the results**

1. Go to the start menu and select Chrome
Navigating to the application

1. In the address field, type **192.168.100.104** followed by Enter.
2. You should now see the demo app. Scroll down and verify that you see the two reviews we have already entered and stored in persistent state.

Extra credits: Enter a few new reviews and shut the application down using the `docker-compose down` command, then bring the application up again and make sure the reviews are still there.
Clean up

1. Shut down the application. Type the following and press Enter:
   ```bash
docker-compose down
   ```
2. Remove any leftovers. Type the following and press Enter:
   ```bash
   /root/cleanup.sh
   ```

vSphere Docker Volume Service

Docker volumes are a great way of making data persistent across invocations of containers. But the standard, built-in Docker volumes have a serious drawback: They are local to the Docker host where they were created. This becomes a problem when deploying an application to a cluster, such as a Docker Swarm or Kubernetes cluster. Since the volume is local to a host, data would not be accessible if a container failed over to another host.

To address this problem, VMware has released vSphere Docker Volume Service (vDVS). By implementing a Docker volume driver that's backed by a vSphere Datastore, volumes can be accessed across different Docker hosts as long as they can access that datastore.

Installing vSphere Docker Volume Service

Installing vSphere Docker Volume Service takes only a couple of minutes. This process has already been finished in your lab, but we will outline the steps for illustration purposes.

There are two steps to this installation. The first step installs a driver module into the ESXi servers for managing the volumes. The second command installs a volume driver on the Docker hosts. Please note that these commands have to be executed on every ESXi host in the cluster and every Docker host, respectively!

**Don't type either of the next two commands! They are for illustration purposes only!**
1. Log on to every ESXi server in the cluster you plan to use and type the following command (after downloading the VIB):
   ```bash
   esxcli software vib install -v /tmp/
   VMWare_bootbank_esx-vmdkops-service_0.14.0577889-0.0.1.vib
   ```
2. Log on to every Docker host and type the following command:
   ```bash
   sudo docker plugin install --grant-all-permissions --alias vsphere vmware/
   docker-volume-vsphere:latest
   ```

## Creating a vSphere backed volume

Creating a volume is very simple. The only difference between a standard Docker volume and a vSphere backed volume is that we need to explicitly specify the volume driver using the `-d` option.

1. To create the volume, type the following command and press Enter:
   ```bash
   docker volume create -d vsphere vol1
   ```
2. Docker will print the name of the volume as confirmation that it was created. Let's start a container that uses it! Type the following command and press Enter:
   ```bash
   docker run -it --rm -v vol1:/data busybox
   ```
3. Notice the `-v` option! We are mapping the volume we just created to the directory `/data` inside the container. Let's create a file in that location! Type the following two commands and press Enter:
   ```bash
   cd /data
   echo "HOL rocks!" > test.txt
   ```
4. We've now created a small test file on the vSphere backed volume. Type the following command and press Enter:
   ```bash
   ls
   ```
5. You should see the file you just created. We can now exit the container. The `--rm` option we gave when starting it will remove the container, but keep the volume we created as `vol1`. Let's exit the container by typing the following and pressing Enter:
   ```bash
   exit
   ```
Accessing our file from a different Docker Host

1. On the toolbar, click the PuTTY icon as shown.
2. In the window that pops up, select Docker-CLI-2.corp.local
3. Click Open.
4. In the command window that pops up, type the following and press Enter:
   ```bash
   root
   cd /data
   cat test.txt
   exit
   ```

Log into the second Docker host.

1. On the toolbar, click the PuTTY icon as shown.
2. In the window that pops up, select Docker-CLI-2.corp.local
3. Click Open.
4. In the command window that pops up, type the following and press Enter:
5. Let’s start a container and map it to the same volume. Type the following and press Enter:

   `docker run -it --rm -v voll:/data busybox`

6. Check that the file we created is still there even though we’re running on a different host. Type the following command and press Enter:

   `cd /data`

7. Type the following command and press Enter:

   `cat test.txt`

   You should see the text we added previously when running the container on the other container host.

8. Type `exit` followed by Enter to exit the container.

9. Type `exit` followed by Enter to exit the terminal. The terminal window should close.

---

**Looking at the volume in vSphere**

Let's have a look at what happened behind the scenes in vSphere. You will notice that it's all implemented using standard vSphere components. The volume is mapped to a VMDK residing in a standard datastore. This means that it can be treated just like a normal VMDK for backups etc.

Let’s start by logging into vCenter.
1. On the Windows taskbar (bottom of the screen), click on the Chrome icon (as shown above).
2. In the Chrome window, click vCenter (HTML5) on the toolbar.
3. Once the login page has loaded, check the Use Windows session authentication checkbox.
4. Click the Login button.

Navigating to the vSphere Docker Volume Service datastore

Once the vCenter UI appears, we can navigate to the datastore and inspect it.

1. Select the Datastores icon (as shown above)
2. Expand the RegionA01 datacenter
3. Select the RegionA10-ISCSI01... datastore
4. Click the Files tab.
5. Expand the dockvols folder
6. Select the _DEFAULT folder.
7. Inspect the content. The vol1.vmdk file is what backs the Docker volume we just created.
Conclusion

In this module, we have covered the following:

- The basics of VMware PhotonOS
- Deploying with Docker Compose
- Volumes and stateful components

You have finished Module 3

Congratulations on completing Module 3 and the HOL-1830-01-CNA - Container Basics with Photon OS lab!

You should now have a solid understanding of basic container and Docker concepts, as well as gained some experience deploying containers on a virtualized container host.

You may next want to explore how VMware VIC (vSphere Integrated Containers) adds additional and unique value to container management and operationalization. This content can be found in the following lab:

HOL-1830-02-CNA - vSphere Integrated Containers.

If you have already finished the other Modules, you may now end the lab.

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-1830-01-CNA

Version: 20180125-174806