Course Overview
Course Overview

(Music) Hi everyone. My name is Anthony Nocentino, enterprise architect and founder of Centino Systems. Welcome to my course, Kubernetes Installation and Configuration Fundamentals. Are you a systems administrator that needs to build and administer a Kubernetes cluster? If you do, then this is the course for you. First, we'll start off by introducing you to the Kubernetes and the world of deploying container-based applications. We'll look closely at Kubernetes architecture, including cluster components and networking fundamentals. Then we'll look at how to build your own Kubernetes cluster, and we'll look at installation considerations, such as deploying on premises or deploying in the cloud. Together, we'll deploy clusters in both scenarios in some demonstrations. And we'll wrap it up with learning how we can interact with our cluster at the command line using kubectl and the basics of application and service deployments in Kubernetes. At the end of this course, you'll know how to build a Kubernetes cluster, both in local virtual machines and as a managed service with some of the major cloud providers. You'll learn what it takes to get an application up and running in your Kubernetes cluster. Before beginning this course, you should be familiar with the Linux operating system and administering it at the command line. You should have a firm understanding of TCP/IP-based networking and also understand the fundamental concepts of containers. I hope you'll join me on this journey to learn how to build a Kubernetes cluster and deploying container-based applications in Kubernetes in the course, Kubernetes Installation and Configuration Fundamentals.

Exploring the Kubernetes Architecture

Introduction, Course, and Module Overview

Hello, this is Anthony Nocentino with Centino Systems. Welcome to my course, Kubernetes Installation and Configuration Fundamentals. This module is the Introduction and Exploring Kubernetes Architecture module. In this course, we're going to get you started on your journey into building and configuring your own Kubernetes cluster, whether it's on-prem or in the cloud. So let's start off with a course overview. We're going to kick it off with an introduction, and then we're going to look at the theory and the concepts behind Kubernetes that makes, well, Kubernetes, Kubernetes in the module, Exploring Kubernetes Architecture. Once we have that theory in our brains, we're going to move on into installing and configuring Kubernetes. We're going to look at both on-prem and cloud scenarios, doing a deep dive into how that really all pieces together in building our first cluster together. And then once we have that cluster up and running, I don't want to leave you hanging. We're going to go ahead and learn how to work with our Kubernetes cluster and do some basic operations, like deploying a service and an application into our newly built Kubernetes cluster. All right, In this module, we're going to kick it off with the discussion to answer the question, what is Kubernetes? And once we know that, then we're going to move on to exploring the Kubernetes architecture. We're going to zoom in very closely on the cluster's components and also some basic networking fundamentals.

What Is Kubernetes? Kubernetes Benefits and Operating Principles

Let's start our discussion with answering the question, what is Kubernetes? At its core, Kubernetes is a container orchestrator. What this means is it's Kubernetes job to start and stop container-based applications based on the requirements of the systems administrator or developer. Now, one of the key facets of Kubernetes is workload placement. If I need to deploy a container-based application into a cluster, how do I deploy it? On which servers does it physically live on? Does it need to be co-resident with other services or containers inside of the cluster? If that's the case, we can define that in our code that describes our application, and Kubernetes can manage that for us. Now, speaking of Kubernetes managing things for us, Kubernetes also provides an infrastructure abstraction. As a developer, if I need to deploy an application into production, I really don't want to have to care about which server it lives on or have to go configure a load balancer to send traffic to my application. That's all going to be handled for me under the hood by Kubernetes. One of the other core ideas behind Kubernetes is this concept called desired state. We can define what our applications or our services look like in code, and it's the job of Kubernetes to make sure that our system meets that defined desired state. And so perhaps our application or our system is composed of some collection of web application containers and database containers, maybe some middleware or even a caching tier. We can write the code that describes what our system looks like, hand it off to Kubernetes for deployment, and it's Kubernetes job to make that happen, to make sure that our system is in that defined desired state. Let's look at some of the key benefits of using Kubernetes, and first off is speed of deployment. Kubernetes gives us the ability to deploy container-based applications very, very quickly. And what this enables us actually to do is to get code from a developer's workstation into production fast, and that gives us the ability to absorb change quickly. The speed of deployment really allows you to iterate quickly and get new versions of code out, enabling new key capabilities for your organization's applications. Next up is Kubernetes' ability to recover very quickly. When we define our system and code and we define that desired state, so perhaps a collection of web app containers, and something causes our system to no longer be in that desired state, so perhaps a container crashed or a server failed, well, it's Kubernetes' job to ensure that our application comes back to that defined desired state by deploying new containers supporting our applications, making sure that we have a collection of web app containers up or really whatever resource it is that defines our application or our system's desired state. And then finally, Kubernetes allows us to hide infrastructure complexity in the cluster. And so things like storage, networking configuration, and workload placement are core functions of Kubernetes, and developers don't have to worry about these things when deploying container-based applications into Kubernetes. So , and now that we know what Kubernetes is and the benefits of Kubernetes, let's look at some of the basic operating principles behind Kubernetes. And first up is desired state or declarative configuration. This is where we define our application, or really, our deployment's state and code. We define what we want to deploy, and Kubernetes makes that happen for us. It'll go and pull the specified container images and starts them up and possibly even allocates load balancers and public IPs if that's what's defined in our deployment's code. We write the code describing the deployment. Kubernetes does the work to bring it online in the desired state. Next, controllers or control loops have the responsibility of constantly monitoring the running state of the system to make sure that the system is in that desired state. And if it's not, a controller will try to bring the system into the desired state. And so, for example, if we've defined that we want three web app containers online, it's Kubernetes job, more specifically, a controller's job, to ensure that three web app containers

are online. A controller will start the three replicas up, and later, if one of those fails or goes offline, a controller will create a new web app container, replacing the one that failed. Now there are many different types of controllers available in Kubernetes for various scenarios, and we'll cover them throughout this series of courses. But the key concept here is controllers are what make changes to the system to ensure the desired state. Another core principle is the Kubernetes API. The Kubernetes API provides a collection of objects that we can use to build and define the systems that we want to deploy in code. The objects defined in our code define the desired state of our applications or the systems that we want to deploy in Kubernetes. The Kubernetes' API is implemented and available via the API Server. The API Server is the central communication hub for information in a Kubernetes cluster. This is where we, as administrators and developers, interact with Kubernetes to deploy and manage workloads. And that's also where the components of a Kubernetes cluster interact with each other to understand the current state of the system and to make changes to that state, if needed to ensure the desired state.

Introducing the Kubernetes API - Objects and API Server

So let's look more closely at the Kubernetes API. Behind the scenes, you'll find they're a collection of API objects. These objects are primitives that represent the state of the system, so things like Pods, which represent a deployed container, or nodes, which represent servers that do work in your system. These API objects allow us to configure the state of the system, and we could do that, both declaratively and imperatively, where declaratively, we learned, means we can describe the implementation that we want to have, or we can describe that deployment and how we want our system to look. Imperatively means we sat down at the command line and executed a sequence of commands to get the system to be in the state that we want it to be in. So as we continue along learning about the API server, the API server is a RESTful API that runs over HTTP or HTTPS using JSON. And this is going to be the sole way that we as administrators interact with our cluster, but it's also the only way that Kubernetes interacts with the cluster as well. There are other facets of the system that will be exchanging information, and all of that has to go through the API server. Now when we're communicating to the API server and we're telling it to create things or to change objects into different configuration states, that information is serialized and then persisted into the cluster data store. Now let's look at some of those core primitives, the Kubernetes API objects. These are going to be the building blocks for our deployments inside of Kubernetes. And first up's Pods. What Pods are, are a single or a collection of containers that we deploy as a single unit. This essentially is our container-based application. Then we have controllers. These are the things that keep our system in the desired state. So things like ReplicaSets and deployments are going to become core tools in our toolbox as we build deployments in Kubernetes. Now, services provide a persistent access point to the applications that we deploy in Pods because, as things change under the hood and our Pods get redeployed perhaps by our controllers as they come up and down, those things will be constantly changing. Well, it's the service's responsibility to provide a persistent access point to the applications provided by our Pods. and finally, there's storage. Of course, we'll need to be able to store data somewhere, and Kubernetes gives us some storage objects so we can have persistent storage in our applications. Now this certainly is not an exhaustive list of all the API objects available in Kubernetes, but these are the key players, the core things that we'll build our deployments from.

Understanding API Objects - Pods

So let's look a little closer at each of these core API objects that we just introduced, and we're going to start the conversation with Pods. What a Pod is is a construct that represents one or more containers inside of a Kubernetes cluster. What it really is is those container-based applications or services that you need to deploy into your Kubernetes cluster or your production environment. From a Kubernetes standpoint, it's the most basic unit of work, in fact, it's the unit of scheduling. When we define a Pod, we also can define some of the resources that it requires inside of the manifest that describes our deployment. It's up to Kubernetes to ensure that those resources are available, and schedules that Pods onto those resources in our cluster, and brings our application up and running. Another key attribute of Pods is that they're ephemeral. What this means is that no Pod is ever redeployed. If I deploy an application as a Pod based off of a container today, and that Pod dies, if I go ahead and redeploy that Pod again based off of that same container image, no state is maintained between those two deployments. Once that Pod goes away, it never ever comes back. That new Pod is going to be deployed and provide those services of that container-based application in that Pod deployment. Another key attribute of Pods is the atomicity of a Pod, they're either there or they're not. Now, in a single-container Pod, this kind of makes sense, like a Pod's up and running, if the container dies, well, yeah, that Pod's no longer available. In a multi-container Pod, if I have more than one container deployed inside of a Pod, if one of those containers dies, the entire Pod becomes unavailable, and that Pod is no longer providing the services that it should. Continuing the conversation about Pods, what I want to introduce now is that it's Kubernetes' job to keep your Pods up and running using controllers, but more specifically, it's about keeping your application or your deployments in the desired state. And so to do that, to make sure that things are in the desired state, Kubernetes tracks the state of a Pod. Is our Pod and the containers inside of that Pod up and running? Now of course our Pod could be up and running, but the applications inside of that Pod could be throwing errors, so Kubernetes also tracks the health of a Pod. Is the application inside the Pod up and running? And so Kubernetes can check the health of an application running inside of a Pod with probes. In our deployment code, we can define a probe to check the health of our applications. And so, for example, if we've deployed a web application, we could write a probe that checks the URL and tests to see if the application is responding in an appropriate way. And so if that probe fails, then Kubernetes could understand that and make the appropriate adjustments to that individual Pod that might not be responding in a healthy way, and this could be deleting the Pod and replacing it with a new one. We're going to look much more closely at Pod internals and probes in an upcoming course, but for now, what I want you to understand is what they are and the role that they play inside of deploying applications in Kubernetes. We're going to look at them very closely later on.

Understanding API Objects - Controllers

So now that we know what Pods are and that they can come and go based on status and health, how does Kubernetes manage my Pod state? What's the thing that makes sure that our Pod is up, running, and healthy? Well, that's where controllers come in. A controller defines the desired state for your cluster and the applications deployed in it. It's the job of a controller to ensure that things stay in the desired state. Controllers are exposed to you as

workload resource API objects and are most commonly used to create and manage Pod-based applications deployed in Kubernetes. These API objects will create and configure the controllers needed to create and manage your Pods for you. And they ensure your applications stay in the desired state. So, for controllers to define and manage the desired state of applications deployed as Pods, controllers monitor and respond to the state and health of Pods deployed, ensuring the desired number of Pods are up, running, and healthy in a cluster. And if the state or the health changes, the controller will respond accordingly, trying to make sure that the system stays in the desired state. The first controller that we're going to look at today is the ReplicaSet. You've heard me describe this scenario a couple of times so far in the course where I want to have a collection of web application containers up and running in Pods. Well, that's what a replica set models. It allows us to define the number of replicas for a particular Pod that we want to have up and running at all times. And so if I want to have three Pods of this particular web app running at all times, well, it's the replica set's job to make sure that those three Pods are up and running at all times. If one of those Pods becomes unavailable or unhealthy, for whatever reason, it's Kubernetes' job to delete that failed Pod and deploy a new Pod with the hopes of that new Pod returning the system back to the desired healthy state. When it comes to deploying applications in Kubernetes, you generally won't create replica sets directly, you will create deployments. On the creation of a deployment, it creates a replica set based on what's defined in the deployment. And the deployment manages the state of the replica set, so things like which container image to run and also the number of Pods to create. But more interestingly, the deployment controller manages the transition between two replica sets, and a practical example of this is moving between two versions of an application. When we define our initial deployment, it creates a replica set, let's say running version 1.1 of a container image for our application. We can use a deployment to control the transition to a new collection of Pods, perhaps running version 1.2 of our application. A deployment controller controls the transition and even gives us the ability to roll back if needed. Deployments are core to the success of deploying applications in Kubernetes, and we'll look at them in more detail in this course and in much more detail in upcoming courses. Now, there are many more types of controllers available in Kubernetes, not just ones based on Pods. In future courses, as we continue along in our Kubernetes journey, we'll look at things like nodes, services, and other controllers that provide value to our Kubernetes applications and infrastructure.

Understanding API Objects - Services

So we just learned that Kubernetes controllers can start and stop Pods based on the health of the Pod and changes in application state. Well, what we need to know now is, how does Kubernetes add some persistency to all of that ephemerality? Well, that's where services come in. Services add persistency to the ephemerality of Pods. What a service is is the network abstraction for access to the services that Pods actually provide. And so what Kubernetes does for us is it persistently allocates an IP and DNS name for the application services that are provided by the collection of Pods that we want to front-end with a service. And so, as Pods come and go based on their desired state, under the hood, Kubernetes is going to dynamically update the service with new information for those particular Pods. And so our users or other applications will simply access that front-end IP address or DNS name, and Kubernetes will maintain the plumbing, or the infrastructure underneath, as Pods come and go based on their lifecycle and desired state. And so, as users

or applications access that persistent IP, what Kubernetes will do is update the routing information to make sure that traffic comes in on that persistent IP and is routed directly to the Pods that are and healthy supporting that particular services. We can also leverage services to scale our application by adding and removing Pods based on the demands of that application, and services also provide a load balancing to distribute application load across the Pods providing that application services.

Understanding API Objects - Storage

So, the next question you might have in your head is what about my data? Where does Kubernetes store data for persistent storage in the cluster? And so let's talk about some storage constructs that are available inside of Kubernetes. Initially, Kubernetes had the concept of a volume, which was storage backed by physical media that was directly accessible to a pod. And so as we deployed pods, we defined that we wanted this volume of this type of storage, and those two things were tightly coupled together in a deployment. And so we lost some flexibility in how we could administer pods and how we could administer storage. And so, after that, Kubernetes came up with the concept of a persistent volume. What a persistent volume is is pod independent storage that's defined by the administrator at the cluster level. And so when a pod wants access to that storage, it defines what's called a persistent volume claim. So, in the pod definition, we basically say I want 10 GB of this type of storage, and the pod claims that storage from the underlying cluster. This technique effectively decouples the pod from the storage inside of the cluster.

Kubernetes Cluster Components Overview and Control Plane

So now that we've introduced the basic principles of Kubernetes, let's move on into exploring the Kubernetes cluster architecture and look at what a cluster is actually composed of. And so let's look at cluster components. The first cluster component is the control plane node. The control plane node implements the major control functions of a cluster. It coordinates cluster operations, monitoring, Pod scheduling, and is the primary access point for cluster administration. Next up is the Node, sometimes called the worker Node. This is where our application pods actually run. The Node has the responsibility of starting up pods and the containers supporting those pods. Nodes implement networking, ensuring the network reachability to the pods and services running on worker nodes. Each node in the cluster contributes to the compute capacity of the cluster, and clusters are generally composed of multiple nodes based on the scalability requirements of the workloads deployed. And finally, nodes can be either virtual or physical machines. The control plane node used to be called the master node, and so in some documentation and resources on the web, you will see the term master node used. The modern name for this is control plane node. I'm in the process of updating this series of courses, and we'll update this to control plane node in each course, updating the videos, code samples, and demos. So if you get to a course that I haven't updated yet, you will see this referred to as the master node. This is conceptually the same as the control plane node, and it will be updated as soon as I can get to that course. Zooming in on the cluster's control plane node. The control plane node, which implements the major control functions of a cluster, is comprised of several components, and the first is the API server. The API server is the primary access point for cluster and

administrative operations. This is essentially the communication hub of our Kubernetes cluster. Now, the API server itself is stateless, and so we need a place to be able to store the state of the system, and that's where the cluster store etcd comes in. This has the responsibility of persisting the state of our Kubernetes objects. Next is the scheduler. The scheduler tells Kubernetes which nodes to start pods on based on the pod's resource requirements and other attributes such as administrative policies. And then there's the controller manager. The controller manager has the job of implementing the lifecycle functions of the controllers that execute and monitor the state of Kubernetes objects such as pods. Basically, it has the responsibility of keeping things in the desired state. And we're also going to introduce kubectl here, not really part of the control plane, but it's how we're going to interact with the API server for administrative functions. When we work with kubectl, it's going to interact with the API server for us to be able to retrieve information and also to make those changing operations to help get our system into the state that we want it to be in, such as deploying workloads. It's our primary administrative command line tool for operating our Kubernetes cluster. And now we're going to deal with this right now. I call it kubectl. You can call it kube control, kube cuddle, whatever you want to do, but for this course, we're going to go with kubectl. Now, within the control plane node, let's look even closer at those individual control plane components that we just introduced. We're going to look at the API server, the cluster store etcd, the scheduler, and the controller manager. Now, the API server is central to the control of your cluster. It's core to all operations. All configuration changes pass through the API server. It's a very simple interface leveraging a RESTful API, exposing RESTful operations like GET, PUSH, POST, and so on. Based on the operation that's going into the API server, it's the responsibility of the API server to validate that operation and persist that object state into the Cluster data store, which is etcd, and as we've introduced, etcd is the cluster data store and is used to persist the state of those API objects as key value pairs. Next up is the scheduler. It's the job of the scheduler to watch the API server for unscheduled pods, and then schedule those pods on nodes. When scheduling pods, the scheduler will evaluate the resources required by that Pod in terms of things like CPU, memory, and storage requirements to ensure their availability when placing a Pod on a specific node in our cluster. Further, the scheduler has the responsibility of respecting any constraints that we defined administratively, and so perhaps you want to keep two pods on the same node at all times. That's called Pod affinity. Or perhaps we want to do the opposite, where we want to ensure that two pods are never on the same node. That's called Pod anti-infinity, and the Scheduler will handle that for us. The Scheduler is capable of scheduling pods based on many different other attributes, and we'll look at this more closely in an upcoming course. And then finally, the controller manager. It's the job of the controller manager to execute those controller loops. Those controller loops implement the lifecycle functions of pods ,and thus defined the desired state of the system. It's the job of the controller manager to watch the current state of the system and update the API server to ensure that its heading towards the desired state. The replica set API object is an example of a controller that we will be working with a bunch in this course. This is the controller that is used to ensure that the correct number of pods or replicas are up and running in our deployments. And so it's the responsibility of the replica set controller to communicate to the API server changes in its desired state. And so this could be operations like creating new pods or deleting pods, based on what the running state of the cluster is. And as we've introduced, there are many different other controllers available in Kubernetes, and we'll introduce more in this course and in upcoming courses as we continue our Kubernetes studies.

Nodes

Now that we've introduced the control plane node and its core components, let's take some time to look at what a node does in our cluster. A node is where our application Pods run. The node starts up the Pods and ensures that the containers in those Pods are up and running. Nodes also implement networking to ensure the reachability of the Pods running on the nodes in our cluster. We could have many nodes in our cluster based on the scalability of requirements of the applications that we're deploying inside of Kubernetes, and nodes can consist of either physical or virtual machines. Now, within a node, there are some components that I want to introduce you to, and first up is the kubelet. The kubelet has the responsibility for starting up Pods on the node, then there's the kube-proxy, which has the responsibility for Pod networking and implementing our services abstraction on the node itself, and then there's the container runtime. This is the actual runtime environment for our containers. The container runtime has the responsibility of pulling the container image from our container registry and providing an execution environment for that container image and the Pod abstraction. Now, both the kubelet and the kube-proxy communicate directly with the API server, monitoring it for changes to the environment. And so as Pods are scheduled to this individual node, the kubelet, it will monitor the API server for that information. And if it does have a scheduled Pod, it'll start up the containers needed to support that Pod on the node. Similarly, if there's a networking topology change that needs to be implemented, such as adding a newly-created Pod's IP to a service for load balancing, it's the responsibility of the kube-proxy to monitor the API server and make that modification on the node. Let's look at those individual components of a node in more detail. We're going to look at the kubelet, the kube-proxy, and the container runtime. Now, I do want to call out that all of these components actually will run on all of the nodes in the cluster. What this means is that even on the control plane node, these three services, the kubelet, the kube-proxy, and the container runtime, will run because there are going to be special purpose Pods deployed onto the control plane node providing the control plane services. There's also going to be a kube-proxy providing networking services, and a container runtime as these execution environment for those Pods on the control plane node, and so all of these services will exist on all nodes in our cluster. Looking more closely at the kubelet first, as we've discussed, it monitors to API server for changes. And so, as Pods are scheduled onto individual nodes, it's the kubelet that goes and asks the API server, hey, do you have any work for me? And the API server answers that question, yes or no. The kubelet is also responsible for Pod lifecycle, meaning starting and stopping Pods and its containers in reaction to those modifications or those state changes that are being watched on the API server. In addition to monitoring the API server for changes, it's also the responsibility of the kubelet to report on both node and Pod state. So if a node is reachable and reports its status as ready, then it's available for scheduling of new Pods in our cluster. The kubelets also have the responsibility of monitoring Pod state. Is the Pod up and running? That's all reported back to the API server via the kubelet. If there is a probe used for determining Pod health, it's the responsibility of the kubelet to execute that probe. Now, the kube-proxy has the responsibility of all networking components for nodes, and it's most commonly implemented in iptables. There are other modes for kube-proxy, but for this course, we're going to focus on iptables. Kube-proxy has the responsibility of implementing that services abstraction that we introduced a little bit ago, and the kube-proxy is also responsible for routing traffic to Pods. And so as application requests come into the cluster, kube-proxy has the responsibility of ensuring that those requests land on the correct Pods executing on the nodes in our cluster. And the

kube-proxy also has the responsibility of load balancing, making sure that traffic that's coming into multiple Pods is distributed in an even fashion. The container runtime has the responsibility of downloading container images and starting and running containers. The container runtime in Kubernetes is wrapped up in what's called the Container Runtime Interface, or CRI, and this gives us the luxury of being able to swap out the container runtime, choosing from several different supported container runtimes. Out of the box, we're going to be using containerd. This is the default container runtime In today's version of Kubernetes. There are many other container runtimes available to you, and you can use those in Kubernetes as long as they're CRI, or Container Runtime Interface, compliant. Before Kubernetes version 1.20, Docker was the container runtime used. In version 1.20, Docker was deprecated and will be removed in 1.22, or later. In practice, though, you can still use container images built with Docker in your Kubernetes cluster when using other container runtime as long as they're CRI compliant. In our cluster in this course, we'll be using containerd.

Cluster Add-on Pods

Cluster add-on Pods are Pods that provide special services to the cluster itself, and the primary example of this is DNS. These special-purpose Pods for DNS provide DNS services inside the cluster using the CoreDNS DNS server. The IPs for the Kubernetes service front ending these DNS server Pods and the search suffix of the domain is placed into the networking configuration of any Pods created by the cluster's API server, and so Pods and nodes and services will register their addresses with the DNS server when they're created. And since this is the DNS server used inside of the cluster for its services, it's commonly used for service discovery for applications deployed inside of a cluster. You'll find DNS Pods deployed in nearly every Kubernetes cluster. Other cluster add-ons include ingress controllers, which are essentially advanced HTTP or Layer 7 load balancers and content routers. Also, another example of an add-on Pod is the Kubernetes dashboard for web-based administration of your Kubernetes cluster. Ingress and dashboard Pods are optional, and can be easily added on to any cluster if needed. Network overlays are also classified as cluster add-on Pods. We're going to talk about those in more detail in the next module.

Pod Operations

So, now that we know the basic principles behind Kubernetes in terms of theory and we looked at the components that a Kubernetes cluster is made of, let's see Kubernetes in action and see how this all pieces together. And we're going to start off with some pod operations. Let's say we have a cluster and it's composed of a control plane node and a couple of worker nodes. Using kubectl, we submit code to instruct Kubernetes to create a deployment. And in that deployment, we've defined that we want three replicas of our pod. That request is going to be submitted to the API server, and it's the responsibility of the API server to store that information persistently in etcd. With that then, it's the responsibility of the controller manager to spin up those three requested replicas in that replica set. And so that's going to create those three pods. That request is then going to be submitted to the scheduler. The scheduler is then going to tell the API server that these pods need to be scheduled on nodes that it selects, and that scheduling information is persisted back to etcd.

Which nodes did those pods get scheduled on? Well that's dependent upon the resources requested in the pods and the resources available in the nodes in the cluster.. The kubelets on the nodes will ask the API server, hey, do you have any work for me, and then it's going to start spinning up those pods that were requested by the replica set. Now everything is going along happy, happy, and the controller manager is monitoring the state of the replicas. And everyone's reporting that they're in a desired state. But what happens if a node goes down? Well, that node is no longer reporting state, and the controller manager understands that we're now outside of the desired state in the terms of the number of replicas required for our replica set. So the controller manager is going to submit for a new pod to be created and scheduled, and it's the responsibility of the scheduler to find a node to put that pod on. And so it schedules that new pod to be deployed on a node to bring our replica set back into compliance with the desired state of three replicas. Now, you might be wondering by looking at this picture, why didn't the scheduler schedule that pod onto the control plane node to maybe even out the workload a little bit between the control plane node and the worker nodes? Well, in the default configuration of Kubernetes, the scheduler taints the control plane node. And what this means is is that the control plane node is only going to run pods that are system pod, and so in this case the API server, etcd, the scheduler, and the controller manager. And a user workload is going to be scheduled onto regular worker nodes in the cluster. We can certainly adjust the configuration of the control plane node to untaint the control plane node so that user pods can be started up on it, but that's usually only good for test labs, not good for production systems.

Service Operations

With the basics of pod operations under our belt, let's see how we can extend that into services and provide a network abstraction or a single access point to our end users into the applications running inside of our cluster. Zooming out a little bit, we're not going to focus on things at the node level right now, we're going to look at things from the cluster level. And let's say we're running a deployment that creates a replica set which creates pods inside of our cluster. In this case, we're going to have four web application pods running inside of the cluster. We can expose access to these pods with a service. And in this case our service is going to be HTTP running on TCP port 80. With the service up and running, users in other applications can come along and access are web application pods via this fixed, persistent service endpoint, which is going to be an IP address or a DNS name. And as requests come in, they're going to be load balanced to all of the pods supporting that service. Now, the value of services, aside from simply having a persistent access endpoint into the application when combined with the functionality of a replica set, if a pod becomes unresponsive or is no longer available it's the responsibility of the replica set controller to remove that pod and deploy a new one. Now our users knew nothing of this because behind the scenes that service is going to stop load balancing requests to the failed, deleted pod and will continue load balancing requests across the remaining pods. Once that new pod is up and ready, requests will be load balanced to it as well. This functionality is core to building self-healing applications in Kubernetes.

Kubernetes Networking Fundamentals

We just introduced services and accessing applications on Pods. Now is an appropriate time to take a look at some Kubernetes networking fundamentals. In Kubernetes, every Pod deployed will get assigned its own unique address. And to facilitate for this, there are some networking requirements, or rules, that we have to live by when we build our networked topologies for supporting our Kubernetes infrastructures. And the first rule is, Pods on a node can communicate with all Pods on all nodes in a cluster without network address translation, or NAT. The next rule is, agents running on a node, so things like system daemons and the kubelet, have the ability to communicate with all running Pods on that node. And so, essentially, what we need to be able to to do is to design our network infrastructures such that all Pods on all nodes in a cluster have reachability to each other with the real IP addresses that are on the nodes and the Pods themselves. And we'll look at an implementation of this in an upcoming module when we build a cluster together. So let's look more closely at the networking fundamentals, or how really Pods and nodes can communicate to each other in some various scenarios. And so let's say we have a cluster up and running, and we deploy a multi-container Pod inside of our cluster. Well, those two containers inside that Pod, if they need to communicate to each other, they're going to do that over localhost using namespaces. Now, let's say we deploy some additional Pods. Those Pods aren't self-contained, so they can't communicate over localhost. So they'll communicate to each other over a layer 2 software bridge on the node using the real IP addresses of the Pods themselves, and so that's how they can communicate to each other. Now, let's say we have to reach out onto a Pod onto another node. Let's say the Pods in our first node there at the top need to communicate to that Pod on the bottom. Well, that's going to happen on the real IP address of that Pod, so we'll need to facilitate for layer 2 or layer 3 connectivity between the Pods on the top node there and the Pod on the bottom node there. And so that's going to be really dependent upon our network infrastructure, so we might have to work with our network engineering team to ensure that we have the ability to have layer 2 or layer 3 reachability between the Pods on these nodes. Another common scenario is, if you don't control the underlying network infrastructure is to deploy what's called an overlay network, and that overlay network gives the ability of all of these things to seem like they're on the same layer 3 network and communicating together on an individual Pod overlay network. And the final case I want to look at with you today is external services. Let's say we have some service in our cluster, and we want to expose that to the world. Well, we learned that that's going to be implemented on the kube-proxy, and we stand up that HTTP service, in this case, and we want to expose that to the world. And so that's how we can get external users into our applications on those services, and it's a responsibility for the service to then communicate to the individual Pods that are front-ends. So these are the four core networking scenarios that I wanted to introduce to you today, within a Pod, between Pods on the same node, between Pods on different nodes, and external services accessing things running on Pods inside of your Kubernetes cluster. Now, as you might expect, this is only scratching the surface, and we are going to have a dedicated course just to networking coming up in this Kubernetes series.

Module Summary and What's Next!

Alright, so here we are at the end of the module. We covered a lot. I hope I got you started on your journey of learning Kubernetes well. We learned what is Kubernetes and the basic principles behind its operations. We explored the Kubernetes architecture. We learned a lot about the individual cluster components and how they interoperate to provide those services that Kubernetes provides to us as a platform. And then we looked at some basic networking fundamentals so we can learn how information moves between resources inside of Kubernetes. Well, I think we're off to a good start, and I hope you do too. Now, please join me in the next module where we're going to install and configure our first Kubernetes cluster together.

Installing and Configuring Kubernetes

Module Overview

Hello, this is Anthony Nocentino, enterprise architect with Centino Systems. Welcome to my course, Kubernetes Installation and Configuration Fundamentals. This module is Installing and Configuring Kubernetes. So far, we've looked at exploring the Kubernetes architecture, getting those basic principles underneath our belt. Now it's time to roll up our sleeves and learn how to install and configure our first Kubernetes cluster. First up, in this module, we're going to start off with some installation considerations. Basically, what do you need to know and do before you install Kubernetes? Then we're going to look at the installation overview. We're going to discuss the process of how we actually install a Kubernetes cluster. Then we'll look at where we can get Kubernetes from because, well, we need software to be able to install. Once we have all of that, we'll sit down and we will learn how to install our first Kubernetes cluster together with kubeadm. And then we'll wrap up the module with creating a Kubernetes cluster in the cloud, looking at some managed service offerings from several cloud providers.

Installation Considerations

Let's start this module off with discussing some installation considerations. What this boils down to is where are you going to install Kubernetes? Where are you going to get the service from to deploy your applications? And first up is cloud. And here there are really two major use cases when deploying Kubernetes in the cloud. You can deploy infrastructure-as-a service, virtual machines and deploy a Kubernetes cluster on top of that. With IaaS-based deployments, the networking, the hypervisor, and all of the infrastructure. plumbing isn't something that you have to worry about, but you do have to worry about the operating system, patching it, and installing and maintaining Kubernetes itself as software on those virtual machines. And then there's PaaS offerings, or platform-as-a service offerings. Kubernetes is available as a managed service, and all the big cloud providers provide this. And you can simply go ahead and consume Kubernetes as a service. You don't have to worry about any of the underlying infrastructure or redundancy. So one of the things that you do have to think about when you are going with a PaaS offering, or a managed service offering of Kubernetes, is you lose a little flexibility in the versioning and other features that are available inside of Kubernetes, so there's kind of a tradeoff

there. When you're building your own Kubernetes cluster using virtual machines in a cloud, well, you get to control exactly which version of Kubernetes that you install. But when you do that, you will take on more responsibility in administering the cluster when compared to a manage service or a PaaS offering. So another solution to getting Kubernetes is on premises. You can install Kubernetes on bare metal physical machines, or you can get Kubernetes on prem using virtual machines, and that seems to be a pretty common deployment scenario. And now that leaves us with one giant question. Which one do you choose? Well, that's really dependent on a couple of things. And what I like to think is that it's dependent a lot upon the skill set of the organization and the strategy of the organization. If you and your teams are cloud ready and you're already in a cloud and you're moving forward as fast as you can, well, yeah, it certainly makes sense to deploy in a cloud offering. Now if you want to stay on prem, well, that's fine, too. Again, it goes to the skills and the strategy of the organization. Which one do you choose? Well, take all of these things into consideration when you're working out your installation considerations for Kubernetes. This course is about the installation and configuration of Kubernetes, so we're going to focus a lot on installing Kubernetes in virtual machines. But we are also going to look at some cloud scenarios together at the end of this module. Now let's discuss some additional installation considerations that we want to take into account before we get started deploying Kubernetes. In an earlier module, we looked at Pod networking and how Pods need to communicate with each other. So we do have to take that into account before we install our cluster networking. Do we want to use an overlay network, or do we want to work with a network engineering team to make sure that we have the correct layer 2 and layer 3 connectivity between the Pods and the nodes in our cluster? And we also want to ensure that there aren't any network IP range overlaps with what we're doing in our cluster and in the rest of our network infrastructure. Then there's the question of scalability. Do we have enough nodes in our cluster with the appropriate amount of resources on each node in terms of CPU and memory to support the workloads that we want to deploy into Kubernetes? We also need to take into account that we have enough nodes in our cluster to support the workload in the event of a node failure. And then there's questions about high availability and disaster recovery. If you've noticed so far, we've discussed a single control plane node cluster. There's one control plane node doing all of the work in controlling everything in our cluster. That's certainly not a highly available solution. We'll learn in later courses in this series how we can take that control plane node and have replicas of the API server backed with multiple copies of etcd to provide redundancy for both the API server and the etcd data store. Additionally, we need to take into account for disaster recovery scenarios, ensure that we have backup and recovery of that etcd data store in the event of a catastrophic failure.

Installation Methods

Now, let's look at some installation methods for how we can get Kubernetes up and running, and first up is desktop installations. A desktop installation of Kubernetes is great for development environments or just playing around to learn how the Kubernetes platform works. Primarily, I use Docker Desktop for Mac when I just want to test various things and kick the tires inside of a Kubernetes platform rather than using a full-blown installation or some managed service in the cloud. There is also a desktop version of Kubernetes for Docker for Windows as well. The installation method that's become the standard for installing and bootstrapping a Kubernetes cluster is a tool called kubeadm, or kubeadmin. This is a

package that you can install, and with kubeadm you can bootstrap a cluster and get it up and running in a very fast way. This is the installation method that we will use in this course. And finally, as we introduced a second ago, there are cloud scenarios where you can deploy both IaaS and PaaS offerings In the various cloud providers. There are lots of different ways to get access to Kubernetes and get a cluster up and running. So go ahead and experiment one of these, but in this course we're going to focus on installing on virtual machines in a local on-premises cluster using kubeadm. And we're also going to look at some clouds scenarios in this course as well.

Installation Requirements

When you're installing Kubernetes on bare metal or, like we're going to do in this course, in virtual machines, we have some system requirements that we need to sort out before we get things up and running. And, well, the first thing is you need Linux. We're going to focus on Ubuntu virtual machines in this course, but certainly Kubernetes is compatible with CentOS, RHEL, and many of the other major Linux distributions. Windows worker nodes are available, but that topic is out of scope for this course. You'll need 2 CPUs in your system and also 2 GB of RAM. Now this is the bare minimum to bring up a cluster for a lab. If you're sizing the cluster for a production workload, you will need an appropriate amount of resources to support your workload. Next, you need to ensure that the swap is disabled on your system, and I'll show you how to do that in an upcoming demonstration. In addition to the base system requirements that we just looked at, you'll also need a container runtime. Your container runtime will need to be Container RuntimeInterface, or CRI, compatible. Currently supported is containerd, Docker, and CRI-O. Now, as of Kubernetes version 1.20, Docker has been deprecated and will be removed in version 1.23 or later. For this course, we're going to walk through an installation of containerd and build a cluster with that as our container runtime in our demos. But I'm also going to provide example code for building a cluster with Docker in the course downloads. From a networking standpoint, we've introduced so far the networking requirements for Kubernetes and pod networks, and so we already know that we need network connectivity between all of the nodes in the cluster. Additionally, though, we also need to ensure that each system has a unique hostname and a unique MAC address.

Understanding Cluster Networking Ports

Now I want to introduce you to the ports that are required to run Kubernetes in the event that you need to build firewalls or other security perimeters around these resources. Now, we know that this is what our cluster looks like, and we know that the control plane node provides a collection of services such as the API server, etcd, and so on. We also know that worker nodes need to access the API server. Specifically, the kubelet and the kube-proxy on worker nodes will talk to the API server over TCP/IP. And so now, let's go through each of these cluster components together and discuss the ports that are required and who uses those ports so that you can develop the firewall rules to help secure your Kubernetes platform. The API server, by default, runs on port 6443. Now that's configurable to any port number, but that's the default port. Who needs to talk to that? Well, pretty much anything in the cluster and anything that needs to talk to it externally, whether it's you administratively with kubectl at the command line or any pipeline tools that need to have access to the system. Etcd runs

on 2379 and 2380. Who needs to talk to etcd? The API server does because etcd is where the API server persists its data. If you're running a redundant configuration of etcd, the various replicas of etcd will need to communicate with each other over these ports. And so if you start scaling that out for redundancy purposes, these are the ports that are required for each of those additional etcd replicas. Next is the scheduler, which runs on 10251, and it's used by, well, itself. If you go look and see at what IP and port it's listening on, it's listening on 10251, but only on localhost. It's not exposed to the outside world. And the same goes for the controller manager, which runs on port 10252, and it also is just listening on localhost. And then finally, on the control plane node is the kubelet. This runs on port 10250, and all of the control plane components will need access to it inside of our Kubernetes cluster. Now on the worker node side of the house, there's a kubelet running on each worker node, and it runs on port 10250. And the control plane elements will need access to this kubelet. And now, for something that we haven't introduced yet, the NodePort service. It's a type of service that exposes our services and ports on each individual node in our cluster, and those port ranges are going to be allocated from 30000 to 32767. Who needs access to these ports? Well, anything that would need access to the services published on those ports. We're going to cover NodePort services in much detail in our networking course later in the series.

Getting Kubernetes

All right, so we've gone over the high-level installation scenarios, we've gone through the system requirements, but what we haven't talked about is where do we actually get the Kubernetes software from. Well, Kubernetes is maintained on GitHub, so if you go to github.com/kubernetes/kubernetes, that's where you'll find the living, breathing project that is the Kubernetes software. So, honestly, if you're up for it, you can go ahead and contribute to this project as well. It's a very, very active and vibrant community. In addition to just the raw software that's available on this website, there is a ton, I mean a ton, of deep-dive documentation available for you to learn about how things work at the deepest level. In fact, if you're up for it, you can go read the code itself. Now, for this course, and for most of the production systems that I support, I use Linux distribution repositories, both yum or apt depending on which Linux distribution I'm using. In this series of courses, we're going to use Ubuntu, so we're going to get Kubernetes from an apt repository.

Building Your Own Cluster

So with the software in our hands, it's now time to learn the steps that we need to take to build our first Kubernetes cluster together. And, well, step one is install and configure a container runtime and the Kubernetes packages. Today, in this course, we're going to learn how to install Kubernetes from packages and use containerd as our container runtime. Then once the packages are installed, we need to create our cluster. What this means is, using kubeadm, we're going to bootstrap that first control-plane node and get those critical cluster components up and running, the API server, the controller manager, etcd, and so on. With the control plane up and running, then we need to configure our Pod networking environment. In this series of courses, we're going to use an overlay network for Pod networking in our cluster. Once we have our Pod network up and running, we can then join additional nodes to our cluster for worker nodes that we can use for our application workloads. Now, let's look at

some of the required software packages to get started working with our first Kubernetes cluster. Again, we're going to be using Ubuntu in this series of courses, so this means we'll be using the app package manager for package installation. First up, we'll need a container runtime. We're going to use containerd in this series of courses. But as we discussed a moment ago, I'm going to give you code for both containerd and Docker installations. In our upcoming demo, we'll walk through the process of installing and configuring containerd together. And in the course downloads, I'll provide examples for both containerd and Docker since Docker is going to be around until version 1.23, or later. Next is the kubelet. The kubelet is the thing that's going to drive the work on individual nodes in our cluster. That comes from a package named kubelet. There's also a package named kubeadm, or kube Adam, or kube admin, whatever you want to call it, but it's the tool responsible for bootstrapping our cluster and getting the cluster components up, running, and configured. It's also the tool that we're going to use to join additional nodes to our cluster. And then finally, there's kubectl, or kube control, or kube cuddle, whatever you want to call it again, but it's the primary command-line tool that we're going to use to administer the workloads in our cluster. Now, I do want to call out that you want to install all four of these packages on all nodes in a cluster, regardless of if they're a control plane node or a worker node.

Installing Kubernetes on VMs

Now, let's go ahead and look at the sequence of commands that we need to use to get an installed Kubernetes on some Ubuntu virtual machines. And I do want to point out here that we need to do this on all the nodes that we're going to have in our cluster, both control plane and worker nodes. First up, we need to install our container runtime. And on an Ubuntu system we'll do that with apt-get install containerd. When we get into the demos, we'll have some additional configuration steps that are needed to configure containerd to use the systemd cgroup driver. The next thing that we're going to do here is we're going to add the GPG key for the apt repository where the Kubernetes packages live, and then we're going to execute this series of commands to add the Kubernetes apt repository to our local repositories list. Now, don't worry about writing this all down or taking screenshots, these will be available to you in the course downloads. And we're going to go through this process together in our upcoming demonstration. Now, with that new repository installed, we need to tell apt to update its package list, and we can do that with apt-get update. And then we'll use apt-get install to install the three remaining required packages, the kubelet, kubeadm, and kubectl. Now, here's where we're going to deviate from the normal Linux package installation. What we're going to do here is we're going to mark these packages with apt-mark hold for the four packages that we need to install. And the reason for this is we no longer want apt to maintain the upgrading of these packages. We're going to service these packages outside of the normal security updates for the system so that we can control when we move between versions of Kubernetes independent of security patches being applied. And this is needed because Kubernetes has a defined upgrade process when moving between versions. We're going to cover that process in a course later in this series.

Lab Environment Overview

So we just looked at the code-level detail to install Kubernetes on some Ubuntu virtual machines. Now, let's zoom out a little bit and look at the lab environment that we're going to use in this course. So, here you can see we have one control plane node and three worker nodes, and so that's the topology of the cluster that we're going to construct together in the upcoming demonstrations and use throughout this series of courses. We're also going to have kubectl installed on the control plane node. And so we'll drive all of our work from that particular node when we're working with our cluster. The version of Ubuntu that we're using in the lab virtual machines is Ubuntu 18.04. I've tested all the labs with the current version of Kubernetes 1.21 on Ubuntu 18. As versions of Kubernetes change, I will continue to update the code in the course downloads to ensure that they work with the latest version of Kubernetes. So if you experience an issue, please reach out to me via email or in the course discussion board. I'll check things out and update the code accordingly. Now as for our lab VMs, my virtual machines are VMware Fusion. You can use pretty much any hypervisor you want, as long as you have full network connectivity between all of the nodes in your cluster. Minimum CP requirements stand, so I have two virtual CPUs per virtual machine in this lab environment. And I also have 2 GB of RAM in each virtual machine. I'm a little generous with the virtual machines from a disk space standpoint because honestly, it's kind of a harder thing to change after the virtual machine is configured, and so I'm allocating 100 GB for each virtual machine in this cluster, and we also want to make sure that Swap is disabled on each virtual machine that we're using. Now I have all of the host names set for each of these individual nodes, and each of these node names is in a host file on each virtual machine in the cluster with the corresponding IP address, so that I don't have to rely on DNS for hosting resolution between any of the virtual machines, that's going to be hardcoded inside a host file on each node. Now as for host names, here we go. For our control plane node, it's going to be c1-cp1 for cluster 1-control plane 1, and its IP address in my lab is going to be 172.16.94.10. You can use whatever IP in your lab that you want to use. Now if you're coming from a previous version of this course, this system used to be called c1-master1, so please be aware and update any code accordingly. Any where you see c1-master1 in any future course, go ahead and swap in c1-cp1. I'll be updating the videos and the code in each course in this series as I get to them. Next up is c1-node1, so a pretty easy naming convention there, its IP address ending in 1.11, c1-node2 ending in .12, c1-node3 ending in 13. And so that's our cluster, one control plane node and three worker nodes.

Demo: Installing and Configuring containerd

All right, it's taken us a long time to get here, but it's time for our first demo together. In this demo, what we're going to do is we're going to install some packages together, containerd, kubelet, kubeadm, and kubectl. And with all those packages installed, we're going to look at some systemd units for some of these packages to understand how they operate with the underlying systemd system. All right, so here we are in VS Code, and let's get started with the process of installing the required packages to bootstrap our Kubernetes cluster. In VS Code here, what I'm going to do is highlight the code at the top, and I'll execute that code, and it's going to show the output at the bottom in our environment. So this way, we're able to see both the code that I'm executing and the output at the same time. I've already gone

ahead and deployed the four virtual machines that we discussed in the presentation portion of the course, so I have c1-cp1, c1-node1, 2 and 3 all up and running in my environment here. I also right now have an SSH connection open to c1-cp1, as we can see at the bottom here. And so let's go ahead and start the process of installing and configuring the proper software for our Kubernetes environment. The first thing that we need to do, and we're going to need to do this on every node in our cluster, is to make sure that our swap is disabled. And I've already actually done that for this virtual machine, and so here you can see if I do a swapoff -a, I get no output because I've already disabled the swap file. And so let me go ahead and show you how I did that. I did that by editing the fstab. Highlight this here, run that code at the bottom, and there we can see the contents of our fstab. On the second line here, we can see that the swap is disabled because this line is commented out. And so what you'll want to do here is either delete that line, comment it out, save the file, and then reboot your system, and your swap will be disabled. And with that configuration complete, let's move forward into the installation and configuration of containerd. Now I do want to call out that the containerd installation right now is a little bit more complicated than I think it should be, and I expect that this process is going to become more streamlined as containerd becomes kind of the centerpiece of the container runtimes for Kubernetes. And so what we're going to see here is we'll have to perform some extra steps to get this containerd installation configured for us to use Kubernetes on top of it. And what I expect is that this is going to become more simple, and so I promised to keep this code up to date with the latest method, but this is the most current method for configuring and installing containerd, and so let's walk through that process together. The first thing that we'll need to do is to load some modules, and with lines 28 and 29 here, I'm going to load both the overlay and the br_netfilter module. Now that's the runtime module, that's right now in this configuration, but I also need to make sure that they're set on boot, and we can do that here on lines 31 through 34 with this here doc, and that's going to write the appropriate information into the modules-load.d directory in etcd. So we'll run that code there, and it'll create that file in the appropriate directory to ensure that these modules get loaded when the system reboots. We also need to configure some sysctl parameters. And again, we want to make sure that those persist across reboot, and so we'll grab this here doc here from lines 38 to 42, run that code, and it's going to write the appropriate sysctl parameters into /etc/sysctl.d in the file 99-kubernetes-cri.conf. With that file created, we can then apply that configuration with the code on line 46 here, so sudo sysctl, space, --system. Now those configurations that were in that file are applied to the running system. With those configuration prerequisites complete, both the modules and the sysctl parameters set, it's now time to install containerd. We'll make sure we have an up-to-date package list, and we can do that with the code on line 50 here, sudo apt-get update. Once we have the updated package list, we could install containerd. And the code to do that is on line 51, sudo apt-get install -y containerd. Run that code, and that's going to install containerd for us on this system. Now with containerd installed, we need to apply some configuration specific to containerd, and we'll do that with the code here on line 55 and 56. First up, we'll create a directory for a containerd configuration file to live in, and on line 55 we have mkdir -p /etc/containerd to make that directory. Now on line 56 here, we can use the containerd command to generate a default configuration file. So there we see sudo containerd config default. That's going to generate a default configuration file. We're going to pipe that output into sudo tee and write that output into /etc /containerd/config.toml, and inside that configuration file will be the configuration attributes for containerd. We're pretty good with the default except for one change. We need to change the cgroup driver of containerd to systemd, which is required for our kubelet, and we'll look at how to configure the kubelet in

the next demonstration a little bit later in the course. And what we're going to do here is grab the text here on lines 68 and 69, that's our cgroup driver configuration, and copy that into our clipboard, and then open up the config.toml file in vim, and we're going to look for the string that we have on line 65 there, and it ends in containerd.runtimes.runc. And below that, we're going to paste in this text from 68 and 69 in that file. So let's go ahead and find that information in our config file. So jumping down to the bottom here, I'm going to look for containerd.runtimes.runc. Once I find that line, here we go. I'm going to add a new line and drop in that text that I copied from lines 68 and 69. And so there we can see I added that configuration into the configuration file, and we're going to write that out and save it. With that written out, let's go ahead and restart containerd, and we can do that with sudo systemctl restart containerd to make that configuration change for containerd part of the running configuration for containerd.

Demo: Installing and Configuring Kubernetes Packages

With containerd installed and configured, now it's time to move forward and install the Kubernetes packages kubeadm, kubelet, and kubectl, and the first part of that process is to add Google's apt repository GPG key to our system so that we can trust that repository. The next step then is to add the Kubernetes apt repository to our local repositories list, and we can do that with the heredoc that's on lines 86 through 88. Run that code together, and that's going to configure that local apt repository on our local system here. With that new repository added, we're going to want to update the package metadata information for our system so that we can get the package information from that newly added repository, and we can do that with sudo apt-get update. Now let's take a peek at what's available to us to install from that new repository, and I want to look at the kubelet packages to see the different versions of the kubelet that are available. And I could do that with apt-cache policy and then specifying the package in kubelet. I want to pipe that into head and limit that to 20 lines of output. In the output at the bottom here, we can see the different versions of the kubelet that are available as packages in the repository. So there we see 1.20.2-00, 1.20.1-00. And so those are all the different versions of the kubelet that are available to us to install. And what we're going to do here is, we're actually, we're going to pin our installation to a specific version, and we're going to install 1.20.1-00, and I have that set here as an environment variable on line 98. And what we'll do then is, on line 99, we'll specify the exact version of the packages that we want to install. So there we see sudo apt-get install -y, and then the package name equals, and then we're referencing that environment variable that we just declared, version. We're going to repeat that pattern for kubeadm and also kubectl, specifying the version for each, making sure that they're the same, all on 1.20.1-00. And the reason why I want to do that is because later on in this series, of course, we're going to run an upgrade to a newer version of Kubernetes. And so if you saw there was 1.20.2 is available, so that way I have a version to upgrade to when we get to that course. Let's go ahead and run this code together to get the kubelet, kubeadm, and kubectl installed. Now with our packages installed, let's go ahead and mark all four packages with hold, and we can do that with sudo apt-mark hold, kubelet, kubeadm, kubectl, and containerd. And what that will do is prevent these packages from being updated when someone comes along and updates the system for security updates with apt. Now on lines 104 and 105 here, I have the code that'll allow you to install the latest and greatest version of Kubernetes. I held us back one version intentionally and also showed you how to pick a specific version. If you want to just install the latest version that's available, you

can use the code that's on line 104 and 105. So now with the requisite packages installed, let's look at the systemd units for a couple of the services that were installed, and first up is the kubelet. I want to look at sudo systemctl status kubelet.service and look at the output that's generated for this particular system to unit. And we can see something interesting here, is that we see that the main process exited, code=exited, status=255 failed with result exit=code. And we also see that the systemd unit's status is activating, rather than active, which would mean that the service is up and running. What's happening right now is the kubelet's actually crash looping because there's no cluster configuration yet. We haven't told the kubelet to do anything specific. And so later on when we get into the next portion of the course we're going to learn how to bootstrap a cluster. And what that's going to do is write some information into a specific location the kubelet's looking for. Well, that information's not available there yet, and so the kubelet is going to be in what's called a crash loop, looking in that specific location for the configuration information that's generated by the bootstrapping process for our cluster. But we haven't done that yet. And so hang on to that thought right now, and a little bit later in the next demo when we bootstrap the cluster, we're going to revisit this and see when the kubelet is up and running and healthy because our cluster has been bootstrapped. The other service that I want to look at is also containerd, just to make sure that that's in good shape before we move forward with our demonstrations. So sudo systemctl status containerd.service, run that code there. We can see it's active and running, so it's up and loaded and in a proper state. So let's break out of this output here. And one final step is just to make sure that both of these services are set to start up when the system starts up. We can do that with sudo systemctl enable kubelet.service. So let's going ahead and run that code. And we'll do the same, sudo systemctl enable containerd.service to make sure that both of those services are set to start up when the system boots. So what we've done right now is, we've installed the required packages to start building, or bootstrapping, our cluster, what you're going to learn about now in the next portion of the course.

Bootstrapping a Cluster with kubeadm

With the software packages installed, it's now time to start the process of bootstrapping or creating a cluster with kubeadm. Kubeadm is a tool that we can use to manage the process of creating our cluster. It's going to walk through various phases, and build and configure our cluster for us, and right now we're going to walk through each one of those steps together so that you know what's really going on during the creation of a Kubernetes cluster. Now, to bootstrap a cluster at the command line, you'll type kubeadm init, and it's going to begin the process of creating your cluster. The first phase is the pre-flight checks. What kubeadm is going to do here for you is execute a series of checks that will help ensure a successful cluster creation. So things like ensuring that you have the right permissions on the system that you're working with. It's also going to verify that you have the appropriate system resources on this box in terms of CPU and memory, and another important pre-flight check that it executes is it checks to see if there's a compatible container runtime on this system, and if it's up and running. If any of these checks fails, kubeadm will report the error and stop the cluster creation process. The next thing that kubeadm does for you is it creates a certificate authority for you. Kubernetes uses certificates for both authentication and encryption. We're going to look at this in more detail in an upcoming slide. In the next phase, what kubeadm does is it generates kubeconfig files for you for the various cluster components of Kubernetes,

so that they can locate and authenticate against the API server. We're going to look at these much more closely in an upcoming slide as well. Then the next phase is generating static pod manifests. Static pod manifests are going to be generated for each of the control plane pods, and what's going to happen here is they're going to be written to the file system and the kubelet is going to monitor that location. If it finds a pod manifest there, well the kubelet's going to start up the pods defined in that manifest, and we're going to look at this also in much more detail in a few moments. And then, after those static pod manifests are generated, kubeadm is going to wait for the control plane pods to start up. So the API server, etcd, and so on, will be started up as pods. Then kubeadm goes on to taint the control plane node. We discussed this a little bit ago when we looked at pod operations earlier in the course. What this means is when Kubernetes schedules pods, it will never schedule user pods onto the control plane node, it will only ever schedule system pods to run on that control plane node. Then, kubeadm generates a bootstrap token used for joining additional nodes to the cluster, and then finally, in its final phase, it will start up any add-on pods, such as DNS and kube-proxy pods. The process that we just described here is the default process if you use kubeadm init with no parameters. This bootstrapping process is very customizable, and you can build more complex or customized clusters using kubeadm command line parameters, or even a YAML manifest configuration file to describe your cluster's configuration. In this module, we're going to learn how to create a cluster with a configuration file, since our cluster is using containerd rather than Docker as its container runtime. In that cluster configuration file, we will specify that we're using containerd as our container runtime, and we'll also define the C group driver for the kubelet, which is systemd. The reason why we're using systemd as our C group driver is because Ubuntu is a systemd based system.

Understanding the Certificate Authority's Role in Your Cluster

It's time now to zoom in on those facets of kubeadm that I called out a second ago, and we're going to start that conversation with the certificate authority. Kubeadm init, by default, will create a self-signed certificate authority for you. And if desired or required in your organization, you can tell kubeadm init to integrate with an external PKI, or public key infrastructure, if that's needed in your organization. The certificate authority that's created by kubeadm init is used to secure cluster communications. It's going to generate server certificates that are used by the API server to encrypt the HTTP stream that is the API server's communication protocol. So that's going to be HTTP over TLS, in other words, HTTPS. In addition to securing cluster communications, the certificate authority is used to generate certificates for the authentication of users and cluster components like the scheduler, controller manager, and kubelet to authenticate the request of the API server. And so we'll have strong authentication methods for users that have to operate the cluster, any of the cluster components, and also the nodes in our cluster since the kubelets will use certificates to authenticate and verify their identity. The certificate authority files live in /etc/kubernetes/pki and will be distributed to each node in your cluster when you join additional nodes to your cluster using kubeadm init. This distribution process ensures that the nodes trust the self-signed CA. For more information on how to customize your kubeadm init process, check out this link here. This article includes the many customizable parameters for kubeadm init, including how to specify an external PKI configuration, amongst many more.

kubeadm Created kubeconfig Files and Static Pod Manifests

Next up is kubeadm-created kubeconfig files. What is a kubeconfig file? A kubeconfig file is a configuration file that defines how to connect to your cluster API server. Inside a kubeconfig file, you're going to find the certificates used for client authentication. And you'll also find the cluster API server's network location. Usually it's going to be its IP address or DNS name. Often included in a kubeconfig file is the CA certificate of the CA that was used to sign the certificate of the API server. This way, the client can trust the certificate presented by the API server upon connection. Kubeadm creates a collection of kubeconfig files used by various cluster components, and those live in /etc/kubernetes. Let's take a look at each one of those now. The first kubeconfig file that's generated is admin.conf, which is the Kubernetes cluster administrator account, essentially a super user inside of our Kubernetes cluster. And we're going to use this to authenticate to our cluster today in our upcoming demo. Kubeadm also generates kubelet.conf. Kubelet.conf is used to help the kubelet locate the API server and present the correct client certificate for authentication. Similar kubeconfig files exist for the controller manager and the scheduler, again, to tell these components where the API server is on the network and also which client certificates to use for authentication to the API server. These kubeconfig files are generated by kubeadm during the cluster bootstrapping process and are used by these components to locate and authenticate the API server. Later on in this series of courses, we'll look at how to create kubeconfig files for individual users that will locate and authenticate to the cluster API server, and we'll also learn how to set appropriate permissions for users. The next stop on the list in our closer look at what kubeadm init does for us is static pod manifests. A manifest describes a configuration, and in this case it's going to describe the configuration of a pod. And so kubeadm init is going to generate static pod manifests for our cluster control plane components that it needs to bootstrap a cluster and get it up and running. Those static pod manifests are going to live in /etc/kubernetes/manifests. And it's the job of kubeadm init to generate a static pod manifest for each of the core cluster control plane components, and that includes etcd, the API server, the controller manager, and the scheduler. So, kubeadm init generates the static pod manifests for each of the control plane pods, and they live in that special directory. It's the job of the kubelet to watch this directory on the file system. And for any static pod manifests that it finds in this directory, it'll start up the pods described in those manifests. And in this process here, it's the control plane pods. This is how a cluster starts up after a reboot as well. The kubelet is started by the systemd daemon on the node on boot. And then on the cluster we'll find the static pod manifests in this directory, and it will start up any of the pods defined in the static pod manifests. And in this case, it's going to be the control plane pods. Now, you might be wondering, why do I need this special configuration to start up these pods? Well, this is what enables the startup of these core cluster components, well, without the cluster being up and running.

Pod Networking Fundamentals

Now, that wraps up the conversation about kubeadm init and a closer look at each one of those phases. Now, let's shift the focus over to the last thing that we need to talk about before we actually stand up our Kubernetes cluster. We're going to talk about Pod networking again. Let's say we're designing a cluster. We have the requirements of the Kubernetes networking model that we introduced earlier in this course. We need to ensure

that we have a single un NATed IP address per Pod and that all Pods have reachability between each other Now, we could, of course, use direct routing, and in this case, we would have to configure the infrastructure underneath our Pods and nodes to support full IP reachability between the Pods and nodes without NAT using real IPs, and sometimes that's just not feasible. Perhaps you're in a cloud scenario or in a scenario where you don't have control over the underlying infrastructure, and that's where overlay networking comes in, sometimes called software-defined networking. With overlay networking, what happens is we get the appearance of a single layer 3 network that the Pods can all communicate on. And some of the techniques that are used to make this happen are tunneling and encapsulation of the IP packets, and that's the responsibility of the overlay network to facilitate for those communications of those packets between the Pods in an unchanged way. Now, some of the overlay networks that are available to us include Flannel, Calico, and Weave Net. These are overly networks, each of which have different capabilities and features that may be interesting to you as you deploy your overline network. In our demos today, we'll be using Calico for our Pod network. When deploying a Pod network, it will provide the IP address management for the Pods deployed. Now, it's imperative that the network range must not overlap with other networks for your network infrastructure, so be sure to work with your network engineering teams to find an appropriate address range to use for your Pod network. For a full discussion of overlay networking, I encourage you to check out this link here. You'll find many more overlay networks available and other networking solutions that you can use for your Pods and nodes so that they can communicate to each other. This discussion here is really just scratching the surface for what you need to know about networking to get a cluster up and running. We have a full course dedicated the Kubernetes networking coming up later in this series.

Creating a Cluster Control Plane Node and Adding a Node

With all of the preliminaries out of the way, it's time to use kubeadm to bootstrap the control plane node in our Kubernetes cluster. The first thing that we need to do is download a YAML manifest that describes our Pod network that we want to deploy, and that's calico.yaml. This is the actual deployment manifest for our Pod network. Inside of that file is where you can specify the Pod network IP address range. We'll look at how to configure that upcoming demo. If this URL changes for the Calico Pod network, I will keep the course downloads updated with the new URL. Next, you'll need to create a cluster configuration file, and there's an easy way to do that with kubeadm. We can use kubeadm config print init-default and then write that out to file. And so here we're using tee to write the output to console and also into a file manifest named ClusterConfiguration.yaml. Inside of that file is the configuration defaults for a Kubernetes cluster. In our upcoming demo, we're going to use this command to generate that cluster configuration file, and then we're going to make some required changes to that file in our lab environment together. Once we have that ClusterConfiguration file, the next thing that we need to do is to execute sudo kubeadm init, kicking off the kubeadm init process that we just walked through together. We need to then pass in our ClusterConfiguration file with the --config parameter and specifying the file location of the configuration file that we just created, ClusterConfiguration.yaml. We then need to specify the --cri-socket of our container runtime, and we're going to be using containerd in this course, and so here you can see the path /run/containerd/containerd.sock, which is the file location of our --cri-socket. If you don't

specify a --cri-socket in the current version of kubeadm, it will default to Docker and will error out. I expect this to change and be updated to autodetect the container runtime in future releases of kubeadm. When you get into the demos, if you decide to use Docker Azure container runtime for your lab, you'll find the kubeadm init process is a bit different. In fact, it's a lot simpler. The reason is a lot of the configurations are autodetected and don't require cluster configuration files or specifying the --cri-socket at the command line with a parameter. But like I said a second ago, I expect the containerd implementation to get a lot simpler, and I'll update the course accordingly. Now, once kubeadm init finishes, all the control plane Pods will be up and running. Kubeadm init will also print out the commands and the parameters needed to join additional nodes to your cluster, and it will also print out how to configure a kubeconfig file for an administrative user. We'll review the join command in a moment, but now let's look at how to work with that kubeconfig file for an administrative user. The following steps will enable you're currently logged in user to perform administrative functions on your cluster. And first what we'll do is we'll create a directory for our kubeconfig file in our current HOME directory, and then what we'll do is we'll copy the admin.conf kubeconfig file from /etc/kubernetes into our HOME directory. And then what we'll do is adjust the permissions on that file so that we can access this with our current user. This kubeconfig file allows the current user to perform administrative functions against the cluster, and inside of that kubeconfig file you'll find the required certificates to authenticate to the cluster and also the network location of our cluster's API server. The final step that we need to take after creating our control plane node with kubeadm init is to deploy a Pod network, and we can do that with kubectl apply -f calico.yaml. This will read the deployment manifest file that we just downloaded, send it into the API server, and then create the Pod network in our cluster. Once the Pod network is finished, then the DNS add-on Pods will be able to start up, and we'll have a fully functioning control plane node. The next thing that we need to do now is to add some additional nodes to our cluster for user workload. Let's check out how to do that. Nodes, or worker nodes, are where your user workloads will run in your cluster. To join a node to a cluster, you first need to install all the required software packages onto your soon-to-be node. So your container runtime, in our case, that's containerd, kubeadm, kubelet, and kubectl all need to be installed. We then used kubeadm with the join parameter to start the process of joining a note to a cluster. Kubeadm join takes some additional parameters, such as the network location of the API server, a bootstrap token used to join a node to a cluster, and a CA cert hash used to trust the certificate presented by the API server. We'll look at some example code for that in a second. When we execute kubeadm join on the node that we want to join to the cluster, what's going to happen is that node is going to download some cluster information and configuration metadata. It's then going to generate and submit a certificate signing request, or CSR, into the API server to generate a certificate that would be used by the kubelet on the node we're joined into the cluster, and this is used by the kubelet to authenticate to the API server. The CA is going to automatically sign that certificate signing request, and then kubeadm join is going to download that certificate and store that on the file system of the soon-to-be node. That certificate is going to live in /var/lib/kubelet/pki on the node. Kubeadm join is then going to generate a kubeconfig file named kubelet.conf, and that kubeconfig file is going to live in /etc/kubernetes on the node. In that kubeconfig file is going to be a reference to that new client certificate and also the network location of the API server that we want to authenticate against for the cluster that this node is joining. This process is called TLS bootstrapping. The code for kubeadm join looks like this: kubeadm join and then the IP address and port of the API server, which in our lab is 172.16.94.10 on port 6443. If you're using a different network address, please update that here. The next parameter is the

bootstrap token. This effectively is a time ticket or password for joining a node to a cluster. And then the next parameter after that is the CA cert hash. The CA cert hash here is used to establish a certificate chain of trust between the node joining a cluster and the certificate presented by the API server for HTTP requests during the join process. Recall a moment ago I told you that the output of kubeadm init when we create our cluster will print this command and the required parameters to the screen so that you can use that to join additional nodes to your cluster. In our upcoming demos, I'll show you just that, and I will also show you how to print the bootstrap token, the CA cert hash, and also the entire join command on demand so that you can get that information when you need it outside of that initial cluster creation process.

Demo: Creating a Cluster Control Plane Node

All right, let's get into a demo and look at how we can create our first Kubernetes cluster together. Once that's up and running, we're going to deploy a Pod network, and then we're going to look at those systemd units again to see what changed from the last time when we installed the kubelet and how it's going to react differently now that we have a cluster up and running because of the static Pod manifests generated by kubeadm init. Then we're going to look at those static Pod manifests for each of our control plane Pods, and then we're going to join a couple of nodes to our new cluster. Here we are logged into c1-cp1, and let's go ahead and begin the process of creating our first Kubernetes cluster together. Now this process that we're going to go through is for containerd. If you're interested and still need the Docker demonstrations, those are available to you in the course downloads here in the docker directory. This demo that we're going to walk through together will be using containerd. Now the first part that we need to go through for creating our cluster is to download the deployment manifest for the Calico Pod network. And have the code to do that here on line 11, and so we'll use wget to retrieve the calico.yaml file from the Calico website. Inside that manifest is a setting that describes the Pod network IP range, and that setting is CALICO_IPV4POOL_CIDR. So let's grab this string here and put that in our clipboard and then open up the file that we just downloaded, calico.yaml, and find that setting inside the manifest. Now what we see here is the value for that field is 192.168.0.0/16. All Pods are going to be allocated IPs from that network range, and so we want to make sure that that network range doesn't overlap with other networks in our infrastructure. If it does, you can set that value here. We're going to leave that as default. And so let's go ahead and break out of this file and get back to our console. Moving forward in the demo, the next part is creating a kubeconfig init configuration file. That configuration file is going to define the settings of the cluster that kubeadm is going to build for us. We can generate a default file with the code here on line 24, kubeadm config print init-defaults. We're going to take the output of that and write that into a file named ClusterConfiguration.yaml. Let's run that code together and review the output of a default ClusterConfiguration file. Scrolling up a little bit, in this output here, we can see, I do want to show you that there is one error that's going to be thrown. In this current version of kubeadm that we're using for this demo, it's going to look for Docker, and, well, Docker is not installed, and so that's an active bug that will be fixed in a future kubeadm installation, so we can safely ignore that warning here. Looking inside of the output of this ClusterConfiguration document, we see LocalAPIEndpoint, advertiseAddress: 1.2.3.4 That's the IP address of the API server, and so we're going to want to update that to our IP address of our control plane node, which in our cluster is going to be 172.16.94.10

or whatever IP address you're using in your lab. The next thing I want to call out is in nodeRegistration criSocket. There we see /var/run/dockershim.sock. We're going to want to update that from the default, which is Docker, to the container runtime that we're using, which is containerd. One other element that we're going to add to this document that's not in the configuration yet is defining the cgroup driver for the kubelet and setting that to systemd, and so we're going to add some configuration there in a moment. Scrolling down a little bit further in this file, we can see Kubernetes version is v1.20.0, and we're going to want to update that to v1.20.1, which is the version of the Kubernetes packages that we installed in the previous demo. And so let's go ahead and begin the process of updating those four elements together. Now I've written some code here that will swap out those values we've set for use that we'd get through this process in a little more streamline of a fashion. And so on line 34, here I have sed that's going to swap out the advertiseAddress from 1.2.3.4 to the actual address of our control plane node, which is 172.16.94.10. Let's go ahead and run that code to make that change. Moving forward to the next update that we have to make, we're going to change the CRI socket to point to containerd, which is the contain a runtime on our node here. And so there on line 38 we have criSocket: \/var\/run\/dockershim\.sock, and we're going to change that to \/run\/containerd\/containerd.sock. Run that code there to make that update and move forward to the next change that we need to make to our cluster configuration, which is to set the cgroupDriver to systemd for the kubelet. Now this configuration doesn't exist in the default document that's generated by kubeadm, and so we're going to add that to our ClusterConfiguration file with this here doc here starting on line 42. And in that code there, you can see on line 46 cgroupDriver is set to systemd for the kubelet. Run that code there to append that to the file that we're working with. Now let's hop into that file to review those changes and make that one final change for our Kubernetes version. Scrolling down in the output here, we can see that the localAPIEndpoint advertiseAddress is now 172.16.94.10, which is the IP address of our control plane node. We also see that the criSocket is now /run/containerd/containerd.sock, and so that update was successful to the configuration document. Going down a little bit further, let's go ahead and change the Kubernetes version from v1.20.0 to v1.20.1. Now remember this is going to be whatever version of Kubernetes that you installed into the previous demonstration, and you want to make sure that those match. So if you move ahead in a version, please be sure to update that here. Scrolling down a little bit further, there at the bottom we can see our kubelet's configuration defining the cgroupDriver as systemd. Let's save this file out and move forward into the next part of our demo where we bootstrap our cluster together. And so on line 58 here I have sudo kubeadm init and then the parameter --config=ClusterConfiguration.yaml. That's the configuration document that we just built up together. We're also going to specify the CRI socket and point that at containerd. Let's highlight all of that code and run that together and begin the process of bootstrapping our cluster. Now I'm going to speed this process up a little bit so we don't have to wait for the whole deployment process, and then we're going to review the output together. All right, with that finished, we can see our command prompt is returned. Let's scroll to the top and review the output of that cluster initialization process together. So in the output here, we can see we're using Kubernetes version 1.20.1. The next phase is preflight checks where that goes through those preflight checks that we talked about during the presentation portion of the course. In the next phase, it creates the CA for us, or that certificate authority, and then goes through a series of commands to generate certificates for each one of the control plane Pods. Then it goes into the kubeconfig phase generating and writing out the kubeconfig files

for admin.conf, kubelet.conf, controller-manager.conf, and scheduler.conf. Moving forward, it then creates the static Pod manifests for each of the control plane Pods. There we see kube-apiserver, kube-controller-manager, and kube-scheduler. It writes those all into /etc/kubernetes/manifests. After the control plane Pod's static Pod manifests are generated, it also creates a static Pod manifest for etcd. And then at that point in time, once all the control plane Pod's static Pod manifests are in place, it goes into a wait state waiting for all those static Pods to start up, and so there we see wait for control plane. And then a few seconds later we see all control plane components are healthy after 24 seconds here in my case. Moving forward into the output here at the bottom, we can then see it creates the add-on Pods. So there we see CoreDNS is created and kube-proxy. Now the most crucial output in all of this here is seeing the output that says your Kubernetes control-plane has initialized successfully! And so when you see that, you know your control plane node is up and running. Now we see some code here that describes how we can connect to our cluster as a user, and that's the code that we reviewed in the presentation portion that will copy the kubeconfig file for admin.conf from /etc/kubernetes into our HOME directory and then sets the permissions on that. And then moving forward, at the bottom there, we can see the join command that we'll use in an upcoming demonstration on how to join a node to the cluster, and we see kubeadm join and then the IP address of our API server, which now is populated with the correct address 172.16.94.10. And so with our control plane node up and running, let's move forward and execute those commands that we need to access our control plane node. And on line 68 here, we can see mkdr -p. In our HOME directory, we're going to create a hidden directory .kube. Run that code and then copy /etc/kubernetes/admin.conf into that directory that we just created and write that into a file named config. That's going to copy that admin.conf kubeconfig file into our user's home director so that we can use that to connect to our Kubernetes cluster. And then next, we're going to change the permissions on that so that our regular user can access that file. With our cluster up and running, we can now deploy our Pod network, and we can do that with kubectl apply -f calico.yaml. Run that code there. What that's going to go ahead and do is create a collection of resources that define the Pod network in our cluster forest. And so now what's happening is a bunch of Pods and resources are being created to implement that Pod network within our cluster. And now let's take a peek at what's happening inside of our cluster, and we can do that with kubectl get pods, and we're going to get pods across all namespaces. Well, really haven't discussed what a namespace is yet, but it's a way for Kubernetes to organize resources, and we're going to focus on some system Pods right now, which are things like the control plane Pods and the Calico Pods that we just deployed. In the output at the bottom here, we can see a collection of Pods that are in different statuses. Let's review some of the output together. And so we see our control plane Pods, etcd, apiserver, controller-manager and scheduler Those are all up and running because we just completed that cluster initialization process. We also see kube-proxy is up and running on this node. That's going to implement service networking on this individual node. We then see our coredns Pods are in the status of ContainerCreating. So what that means is those Pods are out pulling the container images to start those Pods up. And similarly, we see two Calico Pods that are in the status of PodInitializing, or creating the Pod, and then also ContainerCreating, most likely pulling the container image down again. If I use the up arrow and execute that command again, we can see that our Pods are still in that same state. And so rather than continuing to iterate and hit the up arrow and press Enter over and over again, I can add the --watch parameter to that command. And what that's going to do is output to the screen the current state of all the Pods in the system. And as the status of those Pods change, it'll be updated and written to

console. And so this way, we kind of track the deployment of the remaining Pods that aren't quite up and running yet in our cluster, specifically our Pod network Pods, or Calico Pods, and also those coredns Pods. A few moments later, we'll find that all of the Pods are now up and running, including our Calico Pod network Pods and also our DNS Pods. We can check the status of our control plane node with kubectl get nodes, so let's go and run that code there, and we'll see that our control plane node, c1-cp1 has a status of Ready. We can see that its roles are currently control-plane and master and it's up and running on version v1.20.1. That's the version of Kubernetes that we deployed together. We have a functioning cluster up and running. Now we still have to join some nodes to the cluster, but before we do that, let's check out some systemd units again. In the demonstration where we installed the kubelet, we saw that the kubelet service was crash looping because there were no static Pod manifests for the kubelet to start. We hadn't initialized our cluster yet. Well, now that we have initialized our cluster, let's check out the status of the kubelet's systemd service one last time. And we can do that with sudo systemctl status kubelet.service. And in the output at the bottom there, we can see that the status of the kubelet is now active (running) because it has a static Pods to start up when the service starts. Let's take a peek at those static Pod manifests that live in /etc/kubernetes/manifests. Run that code there, and we can see we have a static Pod manifest for etcd, the apiserver, the controller-manager, and the scheduler. If we looked inside each one of these static Pod manifests, these are going to describe what that Pod looks like that it needs to start up. And so in this case here, we have the static Pod manifest for etcd.yaml. Looking inside here at this code, we can see the various configuration elements for etcd. Break out of this output and do the same for the static Pod manifest for the API server, and we can see its configuration as well. And so later on in the series of courses, we'll dive deeper into the static Pod manifest configurations for the control plane Pods. One final thing that we're going to look at in this demonstration is where the kubeconfig files live for the control plane Pods on the control plane node, and those are in /etc/kubernetes. And so there we see admin.conf controller-manager.conf, and scheduler.conf, and we also see the kubeconfig file for the kubelet on the control plane node, kubelet.conf. And so with that demo here, we have a fully functioning up and running cluster with a Pod network deployed. Now join me in the next demo where we'll join a worker node to the cluster for some user workload.

Demo: Adding a Node to Your Cluster

Now it's time to join a worker node to our cluster. You'll find a lot of the steps in this process are very similar to setting up a control plane node, in terms of getting and installing software. The key difference is, once everything is installed and configured, we'll use kubeadm join to join the node to our existing cluster. And so here I have a session open to c1-node1, and let's get started joining this node to our cluster by ensuring that swap is disabled. And so on line 7 here, I have the code to do just that. Swapoff -a, if we get no output, we know the swap is off. And if we need to disable that swap, we'll hop into our fstab, and ensure that our swap entry is either commented out or removed from the file, and there we can see it's commented out. Let's hop back out into our script here, and move forward in configuring this particular node. We're going to start off with installing our container runtime, which is containerd, which is going to require us to load the overlay in br_netfilter modules at runtime here, and also execute this heredoc to make sure that those modules load when the system boots. Moving forward, let's go ahead and create the

required sysctl parameters for our container runtime with the heredoc on lines 31 through 35. Run that code there to save that into the sysctl.d directory, so that's available on reboot, and then on line 39 here, we'll execute sysctl --system to load those settings now in our current running system. With those prerequisites done, we can now install containerd. So I'll issue an apt-get update for that, and then execute sudo apt-get install containerd, to install the containerd container runtime. With containerd installed, we'll go ahead and move forward with configuring containerd. So just like we did on the control plane node, we'll need to make a directory for the containerd configuration in /etc/containerd. And then we'll use containerd to generate a default configuration file with sudo containerd/config default, and then pipe that output to tee, and then create the configuration file in /etc/containerd/config.toml. Now we need to make a modification to that configuration file to ensure our container runtime is using the SystemdCgroup driver. I'm going to copy the code from lines 61 and 62, and then hop into the containerd configuration file, config.toml, and we're going to look for that line that ends in containerd.runtimes.runc, and paste in this code in that location. So let's go ahead and find that in our configuration file here. We'll add a new line, and paste our configuration in, and get that in a proper format, and write that configuration file out. And with that config, we will then need to restart containerd with sudo systemctl restart containerd to affect that change, to make sure that we're using the SystemDCgroup driver within containerd. And so now we can install our Kubernetes packages, and we start with adding the gpg key for Google's app repository, and then adding the app repository to our local repositories lists. Once that's added, we'll then update our package list with apt-get update, and get a quick peek with apt-cache policy kubelet to see all the different versions of the kubelet that are available. And we're going to go with 1.20.1, which is the version that we used on our control plane node. And with that, we can go ahead and install our Kubernetes packages. So I'm going to set the version environment variable, just like we did previously, and then we're going to install the kubelet, kubeadm, and kubectl, and we're going to pin that to the version 1.20.1-00, which is the version of the package that we want to install. With the package installation finished, let's use apt-mark hold to put a hold on the kubelet kubeadm, kubectl, and containerd. So I'll run that code on line 92 here to do that. If you want to install the latest version of Kubernetes, I have the code here on lines 96 and 97 to do just that, but make sure that you match the version that you're using on your control plane node. Moving forward, let's see the status of our kubelet with sudo systemctl status kubelet.service, and we'll find that the kubelet is crash looping right now, because we have not yet joined this node to our cluster. We'll break out of the output at the bottom, and we'll also check the service status for containerd with sudo systemctl status containerd.service. We can see that the containerd service is active and running, and so that's in good shape there to move forward. We'll want to make sure that both the kubelet and containerd are configured to start up when the system reboots, and we can do that with sudo systemctl enable kubelet.service, and we'll do the same for containerd. And so now it's time to join the node to the cluster. We have all of the software installed. We need to join the node to the cluster now, and let's hop back out onto c1-cp1, to begin that process. To join a node to a cluster, you need both a Bootstrap token, and also the CA cert hash. You could have copied this information from the output of kubeadm init, and used that information, but let's say that you didn't. I'm going to show you where you can find that information in your cluster. First up is the Bootstrap token. The Bootstrap token is a timed ticket and has a 24-hour lifecycle. So after 24 hours, you have to generate a new one. If we do a kubeadm token list, it will print any active Bootstrap tokens to screen, and you can see here, the only Bootstrap token, and that it has one hour left, as indicated here in the TTL column. If you have no Bootstrap tokens, you can generate a new

one with kubeadm token create. And here's our new token in the output here. Again, we see that warning about Docker not being installed. This is safe to ignore, and it will be cleared up in future versions of kubeadm. The other key piece of information that you need to join a node to a cluster is the CA cert hash, and so on line 24 here, I have the code that goes and extracts the CA cert hash from actual CA certificate in /etc/kubernetes/pki. If I run this code here, it'll print that information to the console. So there we it begins with 073. Now we could take all of this information and piece it together, and make the join command with kubeadm joins, specifying the Bootstrap token, and then specifying the CA cert hash, or we could use the command on line 129 here, to generate that join command for us. And so let's do that with kubeadm token create --print-join-command. We run that code, and there at the bottom there, we'll get a well-formed join command with the proper parameters and their values, including the location of the API server, the Bootstrap token, and the CA cert hash. So let's grab this text and throw that into our clipboard. And so, with that join command in our clipboard, let's hop back onto our individual worker node. So I'm going to SSH back into c1-node1. Now that I'm on c1-node1, you can see here, I have the text for the join command. I'm going to take that command that I just copied into my clipboard, and paste that over this existing text, because this is a previous token and a previous CA cert hash. And so let's get that onto our screen here, and we'll go ahead and format that so it's a little more readable. And now, let's walk through this join command together. We have sudo kubadm join, and then the IP address of our API server, 172.16.94.10, on port 6443. We then have the Bootstrap token specified with --token, and then the Bootstrap token that was generated when we used the print-join-command command. We then also have the CA cert hash defined there, so we see that beginning with 0735. And so with that, we can then highlight this code and run it, and that will then join c1-node1 to the cluster on c1-cp1. In the output at the bottom there, we could see it's executing the preflight checks, and then it's waiting for the kubelet to perform the TLS Bootstrap process, which we discussed during the presentation portion of the course. And so now, that's all we need to do on the individual worker node. Let's hop back out onto our control plane node, c1-cp1, and check the status of things. If I do a kubectl get nodes now, we can see c1-cp1 is up, running, and ready. We see that c1-node1 is not ready, because what's happening right now is it's deploying the Calico pod network onto that new node that we just added to the cluster, and it's also deploying the kube-proxy pod onto c1-node1. If we do a kubectl get pods --all-namespaces --watch, we can see that it's in the midst of deploying those pods onto the new node that we just joined to the cluster. Let's go ahead and break out of there, and get our console back, and now we should see that c1-node1 is ready, because now the calico pod networking pods are deployed onto the node, as well as the kube-proxy. And so there we can see that c1-node1, its status is up and ready, and it's running version v1.20.1. Now we just installed c1-node1 together. I encourage you, the viewer, to go ahead and install c1-node2 and c1-node3 to complete the build of your cluster. Just repeat the process of the script here in updating the node name as you go through the installation process to the node that you're working on.

Managed Cloud Deployment Scenarios: AKS, EKS, and GKE

So far we've discussed manually installing Kubernetes in virtual machines. That technique can be applied to both on-prem virtual machines or IaaS VMs in the cloud. But what if you want to use them in service? And so I want to take a second here to walk you through some of the managed solutions that are available from the major cloud providers, and first up is Elastic

Kubernetes Service, EKS. This is a managed service offering from Amazon Web Services, or AWS. Here is a link to a getting started guide so that you can go ahead and get started using Kubernetes in the cloud as a service right away. Google Kubernetes Engine, or GKE, is available from Google. And so you can go ahead and consume that service if you want to as well. And here is a link to a how-to document to get you started there. And last but not least, of course, is Azure Kubernetes Service from Microsoft, AKS. Here is a link to a walkthrough document there at the bottom on building your own cluster in AKS. The process for deploying a cluster in a managed solution in any of these cloud providers is effectively the same. Using the command line tooling or web portals, you'll go ahead and authenticate against your cloud provider, you'll deploy a cluster, you'll then download a kubeconfig file to your local machine so that you can use kubectl with a client certificate to authenticate to the API server that's in the cloud for your cluster.

Demo: Creating a Cluster in the Cloud with Azure Kubernetes Service

All right, let's get into a demo. We're going to look at creating a managed service cluster in the cloud in Azure Kubernetes Service. All right, so here we are logged into c1-cp1, and let's get started with the process of creating an Azure Kubernetes Service cluster. The first thing that we need to do is to ensure that we have the Azure CLI client installed. And so let's go ahead and walk through the steps of doing that together. The first part of that process will be adding the repository to our local repositories lists. And then next we will then add the GPG key for Microsoft's repository to our local keychain. Once we have the repo installed and it's trusted, we'll then update our package metadata with sudo apt-get update and then install Azure CLI with sudo app-get install azure-cli. With Azure CLI installed, it's time to log into our Azure subscription. If you don't have an Azure subscription, I have a link here. We can sign up and get free access to a free Azure subscription. And so to log in, we'll use the command az login and run that code. That's going to then initialize what's called a device login. I'm going to log in to my Azure account off screen, and in a moment I'll get the command line back, and I'll know that I'm authenticated to my Azure subscription. Now that I'm successfully logged in, I'm going to make sure that I'm pointing at the right subscription in Azure. So I'll use az account set --subscription. I'm going to point it at my Demonstration Account subscription. And so now with the tooling installed and logged into our current Azure subscription, I can then begin the process of deploying an AKS cluster. And that starts off with creating a resource group, which is a way to organize resources in Azure. And we can do that with the command az group create. I'm going to give it a name, Kubernetes-Cloud, and a location which is an Azure region, and that's going to be centralus for our demo here. So I'll run that code to create my Azure resource group. The provisioningState comes back as Succeeded, so we know that that was successful. Moving forward, let's take a look at the various versions of Kubernetes that are available to me in AKS. And I can find that out with the command az aks get-versions, and then specify a location as centralus, and modify that output into a table format. And like we discussed in the presentation portion of the course, this kind of goes into your cloud selection process because the versions of Kubernetes that are in the cloud are defined by your cloud provider. And so here in the output, we can see the various versions of Kubernetes that are available for us to consume. And so, after that, it's then time to create our Azure Kubernetes cluster. Now you can do that with the command az aks create, specifying our resource group that we just created, Kubernetes-Cloud. We're required to have the generate-ssh-keys switch. And then I'm going to give that cluster a

name, which is CSCluster, and define the node-count as 3. So, we'll highlight that code and run that code together to start the deployment process of our AKS cluster in the cloud. Now, I'm going to speed up the video here until the deployment is finished. All right, so our deployment's finished. We can see that the provisioningState is Succeeded. Let's move forward in the demo. If we didn't have kubectl installed on the system that we're working on, we can use the command az aks install-cli to download kubectl and install that on the local system. But we're on c1-cp1 which already has kubectl installed. And so the next part of the process is going to be getting the credentials from AKS onto our local system so that we can locate and authenticate to our cluster. And we can do that with az aks get-credentials and then specifying the resource group, which is Kubernetes-Cloud, and the cluster that we just created together, which is CSCluster. Now we run that code there. That's going to download the kubeconfig file for the cluster and then merge that into our local kubeconfig file in our current user's home directory. And what that will do is give us two different cluster configuration contexts in our local system. And so, to look at those we can use kubectl config get-contexts. And at the bottom here now we have two cluster contexts available to us. And so we can use a cluster context to tell kubectl which cluster to send commands to. And so here you can see the current cluster context, as indicated by the asterisk, is our AKS cluster. And that second cluster in there is our local kubeadm-based cluster. And so any commands that we execute right now will be sent to our AKS cluster. That AKS cluster context was added when I downloaded the credential from AKS. If we need to switch a cluster context, I can use the command kubectl config use-context, and then specify the context name or the cluster name, which is CSCluster for our Azure cloud. But that's already been set when we merged that configuration in. Now, just to make sure that we're pointing at the right cluster, we can use a command like kubectl get nodes because we know that cluster topology is going to be slightly different than what we're pointing at locally. And so, if I do kubectl get nodes, we can see I have three worker nodes in this cluster. What we don't see here in the output is the control plane node. That's abstracted away for us in Azure Kubernetes Service. What we do see here is just the worker nodes that are supporting our user workload. And if we do a kubectl get pods --all-namespaces, we also won't see any of the control plane pods, like the API server, etcd, the controller manager, and so on. We do see things like coredns, kube-proxy, and some additional pods that are used by AKS to report metrics. When we're all done with our AKS cluster and we want to point back to our local kubeadm-based cluster that we built together, we can switch that cluster context by specifying kubectl config use-context, and then that context name, which is kubernetes-admin@kubernetes. So we'll switch back to our local cluster, and to confirm that we'll do a kubectl get nodes. And there we see our local cluster's nodes. We see the control plane node and then the three worker nodes that are a member of that cluster. I also have commented out here the command to delete this AKS cluster if you need to get rid of it from your subscription. And you can do that with az aks delete, specifying the resource group name and then the cluster name.

Module Summary and What's Next!

All right. So here we are at the end of the module, and we certainly covered a lot. We've looked at some installation considerations and what you need to know before you install Kubernetes. Then we looked at an installation overview and talked about where to get Kubernetes and then installed our first cluster together with kubeadm. We then went on

and discussed managed cloud solutions and did a deployment in Azure Kubernetes Service. Well I hope you enjoyed that module. Why don't you join me in the next module where we'll start working with our Kubernetes cluster?

Working with Your Kubernetes Cluster

Module Overview

Hello. This is Anthony Nocentino with Centino Systems. Welcome to my course, Kubernetes Installation and Configuration Fundamentals. This module is Working with your Kubernetes Cluster. So let's check out where we've been so far in the course. We started off with the introduction. We looked at what Kubernetes really is in the module, Exploring Kubernetes Architecture, we built our first cluster together in Installing and Configuring Kubernetes. Now it's time to sit down and roll up our sleeves and start working with our Kubernetes cluster. We're going to break down this module into two big chunks. We're going to start off with using kubectl to interact with our cluster, and then we're going to look at some basic application deployments into our cluster.

Introducing and Using kubectl

Kubectl, kube control, or kube cuddle, take your choice, it's the primary CLI tool for controlling workloads in your Kubernetes cluster. Now, what we're going to use is kubectl to perform operations against our cluster. Basically, we're going to create, read, update, or delete pretty much any kind of resource in Kubernetes. Now, remember, in Kubernetes, everything goes through the API server, and so kubectl is your primary way to interact with the API server. And so any time you need to make a new thing or query something that exists or make a modification to something, this is the primary CLI tool for doing that. You're going to perform operations on resources, the Kubernetes API objects. Using kubectl is how you're going to manipulate objects like Pods, deployments, services, and others. And so we're going to perform operations against resources. And then finally, the other facet that we're going to look at today in this module is output. If there's output from the commands that we're executing, then we can define the format that we're going to get that output in. And so perhaps we want to have more detailed output with additional attributes exposed or we want to print output as a particular type like JSON or YAML. We can do this all with kubectl. And we're going to start our deep dive into how kubectl works with the operations that you can perform, or really, what do you want to do? And so let's jump right in and talk about the core operations that you'll likely use at the command line with kubectl every day. First up is apply or create, Apply and create are the primary operations for sending deployments and the creation of resources to the API server. Run allows you to start a single or bare Pod when it's not managed by a controller and then also specifying the container image at the command line, so basically, bootstrapping the most basic Pod configuration. One of my favorite commands is explain. This gives you the documentation for API resources, and what this does is it shows you the documentation for a particular Kubernetes API object or resource listing the description and the fields needed to construct that API object. This is a very valuable command when you're working at the command line. Kubectl combined with delete

will delete a specified resource. And then there's kubectl get. What kubectl get will do is display basic information about the specified resource type. And so far in this course, we've looked at nodes and Pods in our cluster, and we're going to use this plenty more as we get into more advanced topics. Next, let's look at describe. Kubectl describe is used to display very detailed information about a particular resource, and it is extremely valuable in troubleshooting scenarios with Kubernetes resources, and we'll look at it in great detail in an upcoming demo. Next up is kubectl exec. Exec allows you to run a command inside a container on a Pod, and so this is very similar to docker exec. And then finally, kubectl logs. Kubectl logs allows you to view the logs that are written to stdout from a container running inside of a Pod. And so this is very valuable for troubleshooting issues with your applications that are running inside of containers inside of Pods. Now this is a short list of what I think are the most critical kubectl operations to get you started. There is a huge list available to you here at the documentation page for kubectl on the Kubernetes website, and I strongly encourage you to check that out there. When working at the command line, we're going to combine kubectl with an operation like the ones that we just introduced with a resource. Basically, what do you want to perform that operation against? And we've introduced things like nodes and Pods and services. And honestly, there are many, many more objects available inside of Kubernetes that we can work with, and so that's how we go ahead and specify what type of resource we want to perform the operation against. Now, here you can see in parentheses an alias to represent that particular type of resource, so nodes, no, Pods, po, services, svc, and that's a good way so that we can get real quick at the command line executing these commands. Now there's a huge list of resources that are available, and you can certainly check that out at that link there, but I'm going to show you some techniques at the command line where you can discover the resource names and the resource aliases because I want you to get very proficient at the command line without having to refer back to the documentation to get things done, and we'll do that in an upcoming demonstration together. The final thing that I want to look at with kubectl today is modifying output. We can specify kubectl's output format by adding additional flags to our commands. The first format that I want to introduce you to today is wide. Using the wide option, we can specify that we want kubectl to output additional information about our Kubernetes objects that have been deployed in our cluster. We can also output our Kubernetes objects as YAML or JSON. YAML and JSON formats are at the core of how Kubernetes describes things declaratively, giving us the ability to describe our configurations in code. We can use kubectl to output YAML or JSON, and this is a very valuable way to get configuration data out of our cluster and describe the resources that have been deployed in our cluster. We can persist this to file and exchange it with other systems, down-level environments, or developers, if we need to. One final option that I want to introduce you to today here is a dry-run. When combined with the yaml output modifier, you can use this to generate YAML for resources that you want to create in your cluster. But dry-run doesn't create the object in the API server; it just out puts the YAML for the object. And so this is a great tool for quickly generating YAML for resources that you want to create, so things like deployments, services, and more, and we'll see this in action later in the module. If you want to dive deeper into how the output options work, check out this link here.

A Closer Look at kubectl

So let's bring all that together and learn how we can use kubectl at the command line. When we're sitting at the command line, we use the command kubectl, we'll specify a command, in other words, an operation, right, that's the thing that we want to do. We'll specify a type or a resource, right, what do we want to do it to, and then, if we need to, we can specify an individual named object. And of course, there's the optional flags that we can append to any command, and so let's look at one command together. If we do something like kubectl get, right, that's the operation, pods, that's the resource, and what if we needed to get the information about a particular pod? Well, we would just append that on. So, kubectl get pods would list all of the pods in the default namespace. Kubectl get pods pod1 will give me the information about just that particular pod, and then I can append on some optional flags. What if I wanted to get that pod's information output as YAML? I would append --output=yaml to get that info. One other command I want to show you guys is kubectl create, right, again, the operation, what I want to create, the deployment, that's the resource. If I need to give it a name, of course I would, nginx, and then I specify an image name, and that's how we could do a basic deployment at the command line using kubectl. Now, I'm going to throw two links at you here, one for the reference documentation and one for a cheat sheet for using kubectl, and I strongly encourage you to check these out. They're fantastic resources, and you might find a small piece of information in there that can help you get through a scenario more efficiently or easily than you're doing it today. And so, go ahead and check out those two links, very good resources.

Demo: Using kubectl: Nodes, Pods, API Resources and bash Auto-Completion

So let's get into a demo where we're going to look at using kubectl. We're going to use kubectl and work with some nodes, pods, and other API resources, and I'm going to throw in a treat here for you. I'm going to show you how to configure bash auto-completion so you don't have to remember all the syntax and shortcuts for kubectl or even resource names. Alright, so here we are logged into c1-cp1. Let's get started with working with our Kubernetes cluster using kubectl. The first command that I want to show you here is kubectl cluster-info, and this is useful for listing and inspecting which cluster you're pointing at in your current context. So I'm going to highlight that code there and run kubectl cluster-info. And at the bottom here, we can see it says Kubernetes control plane is running at https://172.16.94.10 on port 6443. That is the local API server running on c1-cp1 in our kubeadm-based cluster. Now one of the most common operations that you'll use with kubectl is get, and so let's do that together with kubectl get nodes. And what that will do is then print out some critical information about the resource. And in this case, that's going to be a node. And at the bottom there, let's walk through this output together. We have a row of information for each node in our cluster, and let's walk through each one. We have c1-cp1. We see its status is ready, so it's able to take on workload. In this case, on our control plane node, that's going to be control plane pods. We see its role is currently control plane and master. It's been up for about 24 hours, and the version 1.20.1. We see additional rows. For c1-node1, 2, and 3, and those all have a status of ready, meaning that they can take user workload. Now, we can add the output modifier, -o wide, to get additional information about a resource. And so, in this case here, we're going to say kubectl get nodes -o wide, and what that will do is give me additional information about the resource, in this case, our nodes. And so, in addition to NAME, STATUS, ROLES, AGE, and

VERSION, we also have additional fields. So we have the INTERNAL-IP address of the node, so there we see 172.16.94.10 for c1-cp1. In some cloud scenarios, we'll see the external IP populated. Since we're doing this on-prem, that's set to none. We have our OS-IMAGE, so that's going to be Ubuntu 18.04. The kernel that we're running and wrapping off the end of the screen there. We can also see the information about our container runtime. And in our lab here, that's containerd version 1.3.3, and we have a row of information for each node in our cluster. So let's look at the pods that are currently running in our cluster. And if I do kubectl get pods, we get the answer of No resources found in default namespace. Now remember, a namespace is a way to organize resources together. And when we run kubectl get pods, that's going to point at the default namespace. And, well, there's no workload up and running in the cluster yet, so we have no resources found in the default namespace. But there are some pods that are up and running in what's called the kube-system namespace, and that's where the system pods were run in our cluster. And so I can say, kubectl get pods --namespace and then specify kube-system. And then I can see all of the system pods that are running, including the control plane pods, our pod networking pods, and our DNS pods. And so in the output at the bottom here, we see the name of the pod, and then next, we see ready, which tells us if the containers defined in the pod are up and running. Then after that, we have status, which tells us the current state of the pod. Earlier, in a previous demo, when we deployed our pod network, we saw the statuses container creating and pod initializing based on the deployment state of the pod at that point in time and then a transition to running once everything was up running and ready. Next we see restarts, which is the number of times a container restarted inside of a pod, and that that pod was defined about 25 hours ago. We could also combine kubectl get pods with -o wide, and so we can get additional information about a pod. And so let's do that for our pods in our system namespace, so kubectl get pods --namespace kube-system -o wide. Run that code together, and we'll get additional information about a pod. And so we see NAME, READY, STATUS, RESTARTS, and AGE, which is the regular information that we just walked through. Well, we have additional information now. We have IP, NODE, NOMINATED NODE, and READINESS GATES. Let's look closely at IP and NODE. So on the IPs, we see that some of the pods are on the pod network 192.168, and some pods are on our virtual machines network 172.16.94. Depending on the role that those pods play in our cluster, that's what network they'll be attached to. So, for example, our DNS pods will be servicing DNS requests inside the cluster on the pod networks, so those are deployed on the pod network 192.168.00, which we defined in our earlier module when we created our pod network together. Some other pods are on the actual network that our infrastructure is on, so 172.16.94. And so those are exposing services outside of the cluster. So, for example, the API server is available outside of the cluster. It's going to be listening on 172.16.94.10, which is the real address of the control plane node. Additionally, we see four kube-proxy pods up and running. There's a kube-proxy pod running on each node in a cluster. Now recall, kube-proxy has the responsibility of implementing service networking on each individual node. And so there will be a kube-proxy pod on each individual node. So there we see a kube-proxy pod running on c1-cp1, c1-node1, c1-node2, and c1-node3 on the real network IP address 172.16.94. They're exposed to the real network so that they can receive those requests coming in from outside of the cluster and route that information to the correct services and pods running inside the cluster. Now, we can also combine kubectl with get all. And what get all will do is list all current resources that are running in a cluster, and I can also combine that with --all-namespaces. And what that will do is give me every resource that's up and running in my cluster across all namespaces. And so this is a valuable command. They give you a quick view of what's going on in your entire cluster's space. And so the first part of

the output here is pods, and we just walked through that together. And so let's skip forward into the remainder of the output, and we'll see some other API resources defined in our cluster. We can see some services, daemon sets, deployments and replica sets are defined. We'll be diving into each of these in much more detail in some upcoming courses. But for now, the key concept that I want to cover here is using kubectl to display all of the resources that are defined in our cluster. Now, moving forward, let's ask the Kubernetes API server in our cluster about all the types of API objects that it knows about, and I can do that with the command kubectl api-resources. And I'm going to pipe that output into more because there is a large collection of API objects available for us to work with. Now in the output here, we see things like the name of the API object. We see short names or aliases. And so this is the way that we can address a particular object at the command line if we need to. So, for example, if you want to address nodes rather than typing the entire word nodes, we can type the alias or the short name no. After the short name, we see the API version, which is a way of grouping and versioning resources in the API, so we see all of those are on v1. We also can see if an object is namespace or not. So there we see true or false, depending on if that particular object can be in a namespace or not. And then the object KIND at the end in that last column there, we see the different object names. And so, if we need to work with an alias, let's get a quick, simple demonstration of that, kubectl get no, and I'll still get that same output as if I typed out nodes. And so, at the command line, you'll get used to using different various aliases, depending on what your favorites are. Now within that huge list of API resources, one of the quick ways to be able to filter down to find what you want is to type that output in the grep. And so here, we're going to look for the string pods from the output of kubectl api-resources. And again, that's a quick way for me to kind of pare down that list to discover an API resource that I might want to work with. Now if I need to know more about a particular API resource, that's where the command explain comes in, and this becomes one of my more useful commands at the command line when I'm building workloads and constructing workloads. And so let's look at kubectl explain, and we're going to look at the object pod. Now we can put any object type in here from that listing of API objects that we just went through, but we're going to walk through the pod example together. So kubectl explain pod, and I'll pipe that into more. What explain gives you is the documentation about that API object and so that we can learn little bit more about what it takes to construct this type of object. And so here we see the KIND, we see the VERSION, so KIND is Pod, VERSION is v1. This is a v1 pod. We see a description that tells us what it is that we're working with. So a pod is a collection of containers that can run on a host. Then it goes down to the actual fields that are required to construct a pod. And so if we needed to deploy a pod, this is what will be required, the apiVersion, the kind, a metadata, and a spec. If I wanted to dive a little bit deeper to learn about what in the spec is required to describe a pod, well, let's look at that. I can do kubectl explain pod.spec and dive deeper into the description of a pod and learn about what I need to specify when I need to describe a pod in code. And so this, again, is useful for discovering those various attributes. And so if I go down in the output here, you can see that one of the required fields is a container, which makes sense because a pod runs containers, and that's a very useful way to discover how to build that. If I want to even go deeper, I can say kubectl explain pod.spec.containers and run that and dive deeper into that object. And here would be the fields used to describe a particular container. If I go down in this output here at the bottom, we can see things like an image, which makes sense. I'm going to run an image inside of a container to define what container image I want to be started up inside of a pod. So useful information there, again, diving into the documentation to learn about these things at the command line quickly without having to go

look them up either on the web or using some other resources. Now this command, kubectl explain pod --recursive, is a valuable command because what it will do is it will output all of the fields that are part of an API object and march down recursively through all of the fields available in the API object, in this case, a pod. And so what this will do is it will give you the output of the fields for the particular API object, but without the description. So if, perhaps, you forgot a particular field name, this is a valuable way to go and just retrieve that information quickly. So let's break out of that output here and look at one of my other favorite commands that I use very, very frequently at the command line when I'm working with resources defined in my cluster, and that's kubectl describe. And so on line 60 here, I have kubectl describe. I'm going to describe what a node and then a particular node, c1-cp1. Now, I'm describing a node here, but we could describe any other resource. It could be a pod, it could be a service. We're going to focus on nodes at this point in time. Kubectl describe nodes c1-cp1. Let's go and pipe that into more and walk through this output together. So, describe gives you some very detailed information about an API object, and this is extremely valuable when it comes to troubleshooting things that are running in your cluster. And so in the output at the bottom here, we see the name of the resource, in this case, it's a node, which is c1-cp1. The Roles, control-plane, master, and we have some labels and annotations. Labels and annotations are a way for Kubernetes to track and monitor objects that are running in the cluster, and we'll look at those in much more detail in an upcoming course. In addition to that information, we have the creation timestamp when this API object was created. There we see the taint associated with the control plane node. That's going to have the taint of NoSchedule. This is the taint that prevents user pods from running on this node and allows only system pods to run on this node. Moving forward in the output, we see conditions, which describe the current state of the node in terms of things like network availability, MemoryPressure, DiskPressure, and PIDPressure. Moving forward in the output, we also see things like addresses. There's internal IP and then a hostname, the capacity of the node, so the amount of CPU that it's contributing to the cluster, its storage, memory, and so on. We have some additional system information with regards to things like the kernel version, OS image, the operating system that it's running, CPU architecture, and much more, so very valuable deep-dive information about that. We have the current pods that are up and running on the node, and so there we can see the collection of system pods that are running on the control plane node. And so that was kubectl describe. For our control plane node c1-cp1, we can do the same for c1-node1 and dive deeper into this particular node's configuration at the command line here. So there we can see things like its conditions and its status, if it's up and ready, and all of the resources that are available on it, and so on. So at the command line, what we have the ability to do is interact with the API server. And one of the things that we want to be able to do when we're working with Kubernetes is to do these things quickly and discover these things at the command line, so that's one of the reasons why we walk through how to retrieve API objects and their documentation because I want to be able to do that quickly at the command line. Similarly, if I'm working with kubectl, I can just ask kubectl for help, and I can do that with kubectl -h. And then in the output, it has a well-formed organized method of exposing the various operations that you can perform. So there we see basic commands like create, expose, and run, and then going down into more advanced commands for things like deployments and also cluster management. So very valuable information is available in the help. In addition to the basic output right off of kubectl -h, I can combine that with an operation, so kubectl get -h, and then I can get more detailed information about a particular operation that I want to execute. In this case, it's get. And so looking at this, in addition to the normal help that you would see, we also get a collection of

very valuable examples. These examples here are very useful for helping you remember more advanced command lines and texts. The final operation that I want to call out here in the help is kubectl create. This is a command used to create resources in the cluster imperatively, and it's something that we'll be doing very frequently. And as you're working through future demos, be sure to remember that this is here for you as a resource. And now, last in the demo, but certainly not least, here's that treat that I want to share with you. I want to show you that enable bash auto completion for kubectl. And so, let's walk through that process together. On line 71, I have apt-get install -y bash-completion, and that's going to install bash completion on our system. And on my particular system here, we can see that it was already installed. I'm then going to, on line 72, echo in some configuration information into my local bashrc. I'm going to reread that with the source command, and then I'm going to show you at the command line at the bottom here what bash auto completion provides. Now, normal bash completion would do something like this where it would complete the command based on what you've typed at the command line. So I type kube, I hit double tab, I get the auto completion of the three commands that match that string. In that case, kubectl is the one that we want. Now, to extend that, we get auto completion in kubectl for operations. So here you can see if I type g and hit double tab now, it auto completes to get. In addition to operations, it also does that for resources. So if I type po and then double tab, you can see it'll auto complete the pod. If I double tab again, you'll see it auto completes to all of the different resources that are available in my API server that match that string, in this case, pods. And so we'll go ahead type pods there. Also, I can then add something like -- to see any of the different modifiers that are available to me at the command line, so if I double tab now, we can see all the various different modifiers that are available to me at this point in the command line string that I'm building. Let's go ahead and type something like all, and then I'll do a double tab on that, and we can see it's going to limit the list of modifiers down to the modifiers that match the string --all. And so I'll go ahead and auto complete that with -namespaces. Execute that code there. We can see how we can use that to quickly work at the command line to execute commands and discover commands and discover resources that are available in our cluster.

Application and Pod Deployment in Kubernetes and Working with YAML Manifests

So, now that we know how to interact with our cluster at the command line, let's move the conversation to application deployment in Kubernetes. Now, the first thing I want to discuss is imperative configuration. When you're using imperative configuration, you're generally going to be executing commands at the command line one at a time, and you're going to be operating on one object at a time. So, for example, if I want to create a deployment, I can use kubectl create deployment nginx, and specify the image as nginx. I punch this in at the command line, I hit Enter, the command is sent to the API server, and the object is created, but I'm only operating on one object at a time on the command line. Now, similarly, I can operate on other types of objects certainly. I can say kubectl run nginx, and it will create a pod for me running nginx. That's a fine way to manage a system, but if your application stack starts to grow and your configurations and your deployments become more complex, managing each individual object at the command line isn't really a sustainable way to manage or maintain your system. Were going to want to do things declaratively, and this is a core principle behind kubernetes, where we define our desired state in code. Earlier in the course, we introduced the concept of manifests, first when we deployed our pod network, and

second when we bootstrapped our cluster with static pod manifests. Those manifests described those configurations in code, and Kubernetes was able to use them and bring the system up to the desired state as described in those manifest files. And we can do the same for our own applications. We can define our configurations in code using manifests written in YAML or JSON, and feed those into the API server with commands like kubectl apply. In this case here, you can see kubectl apply -f deployment.yaml, and the contents of deployment.yaml will have the description of the thing that I want to deploy inside of Kubernetes. Let's look at our first manifest together, and we're going to look at a deployment manifest. We could have manifests for many different types of API objects available in Kubernetes, but we're going to start our conversation off with deployments. The first thing that you're going to find in any manifest is the apiVersion, and in this case here, since we're using a deployment, it's going to be apps/v1. As the Kubernetes API develops and changes to give users stability as to how API objects are defined and behave, it's versioned. This way, we know a v1 deployment will always look and behave a particular way when we call that in our code. If a newer version of an API is released, and the API objects either change in definition or behavior, that can be a breaking change, and so in our manifest we specify the API version, and this gives us control over which version of an API object we're consuming, and thus adding stability to the objects defined in our manifests. The next thing that you're going to find in a manifest is kind, or the kind of object that we want to define. In this case, we're defining a deployment. Remember in an earlier demo when we did kubectl api resources to list all of the API resources available, those are the objects that we can use here. We'll need some metadata to describe what we're working with here, and in our case, we'll just give our deployment a simple name, and we're going to call it hello-world. Next is the spec. This defines the implementation details of the deployment object. And the first thing that we're going to define is the number of replicas. This is the number of pods that we want up and running for this deployment, and we're going to start off with 1. The selector is a way for a deployment to know which pods are a member of this deployment, and we'll be diving into labels and selectors in much more detail in an upcoming course. And then there's the template. This is the section that's used to define the pods created by this deployment. You'll also hear this called the pod template. In here, we'll have another metadata section, and we'll define some labels. These are matched with a selector in the deployment spec above, and these are assigned to each pod created by this deployment, and that's how the deployment is able to track which pods are a member of this deployment. And then finally, there's another spec. Here's where we will define the containers started by the pods in this deployment, and we define an image to do that. Here we're going to be using a simple hello-world application from the Google Container Registry, and we'll also need to give this container a name, and we're going to call it hello-app. Now, to deploy this deployment, we can save this information into a file named deployment.yaml, and then we'll use kubectl apply -f, and then pass in that file name as a parameter. That will then read the file and feed that manifest into the API server, and then the API server will go and make that happen for us, affecting the desired state of the application in the cluster, and in this case, it's starting up one replica of our hello-world application pod. Now you might be thinking, how am I going to remember all of this? Well, to build YAML manifests for deployments, or really any objects that I want to work with, well, remember in the last demo when we talked about kubectl explain, you can use kubectl explain to quickly look up the fields for an object to help you fill out the implementation details for your manifests. Additionally, we had previously discussed the dry-run output modifier. We can use that to generate basic manifests like this very quickly. Let's look at dry-run in more detail now. Now, we can certainly write the YAML for our

deployment manifest by hand and ensure that we get all of the fields and all the spacing right, but what I want to show you now is a way to generate the YAML needed to create a deployment, or really any API object, on the fly at the command line quickly and correctly. A few slides back, I showed you how to create a deployment imperatively at the command line. We can take that code and its parameters and combine it with --dry-run=client -o yaml, and what this will do is generate the YAML for the API object that you want to create, but it won't send it to the API server for creation. What dry-run client -o yaml will do is write the object YAML to stdout. We can combine that with file redirection and write the output into a file, and so here you can see we're sending the output into a file named deployment.yaml. With this YAML written into the deployment.yaml file, we can use kubectl to send that deployment into our API server for creation or use this output as a starting template for a more complex deployment scenario. I can't stress enough how helpful the dry-run parameter can be to quickly generate correct YAML representation of objects, nearly any object, not just deployments. So I encourage you to use it frequently at the command line to help you get proficient at generating manifests quickly and correctly at the command line. Now it's time to discuss what the API server is actually doing for us when creating a deployment, and we're going to discuss the application deployment process in Kubernetes. And at this point, we're going to bring together a lot of the theory and concepts that we've been going through throughout this course, and so really, this is the time when we're going to bring all of those elements together. And so let's say we have a cluster and we're sitting at the command line with kubectl and we want to deploy an application into Kubernetes and we say kubectl apply. We then pass in some sort of manifest describing the objects that we want to create. So let's say we want to create a deployment. That deployment will create a replica set, and that replicas set will create our pods based on the pod spec in the template, and kubectl is going to send our request into the API server. The API server is going to parse that information defined in the manifest and store those objects persistently in etcd. The controller manager is watching for any new objects that it needs to know about. Since we define a deployment, it's going to start up a controller for that deployment, and that will create a replica set. That replica set is going to create the required number of pods to support the configuration and write that information about the pods back to etcd. Now, the scheduler is watching etcd to see if etcd has any pods that haven't been assigned to nodes yet, and if it finds any unscheduled pods, it will schedule, or in other words, assign a pod to run on a particular node in a cluster. Each pod object is updated in etcd with the assigned node that it needs to run on. Now, no pods have started yet. What we have is the objects for the deployment, the replica set, and the pods with their scheduled node information all written into etcd. So how do the pods actually start? Well, the kubelets on the nodes are watching the API server asking, do you have any work? Do you have any work? And if it finds a pod scheduled for that node, it's then going to send a message to the container runtime on that node to pull down the appropriate container images specified in that pod spec, and start the pod up on that node. If that pod is a member of a service, then that service's information is updated in kube-proxy on that node. This entire process is how pods get deployed inside of Kubernetes. In this example, we described a deployment. Similar processes exist for deploying other types of objects in Kubernetes where the workflow might vary slightly. I just wanted to highlight this example for you today to connect the dots of all the various things that we've discussed throughout this course.

Demo: Imperative Deployments and Working with Resources in Your Cluster

Alright, so let's get into a demo, and in this demo, we're going to deploy some resources using two different techniques. We're going to look at things imperatively and declaratively. Using those techniques, we'll learn how to deploy resources into our cluster, focusing on deploying deployments, pods, and services, and once we have things up and running, we'll look at how we can make changes to existing resources in our cluster, using both declarative and imperative techniques. So here we are, logged into c1-cp1, and let's begin the process of deploying resources imperatively in our cluster. What we're going to do first here is to create a deployment together, and I have the code on line 8 to do just that. And so on line 8, we see kubectl create. What do we want to create? A deployment. where we'd give that deployment a name, hello-world. And then we're going to specify the container image that we want to run in the deployment with the parameter, --image. And so here for the image, we're using a simple hello-world app container image from Google Container Registry, hello-app, with a tag of 1.0. And so when I highlight that code and run that there, and at the bottom we can see deployment.apps/hello-world created. And so what this code is going to do is create a deployment, which will create a ReplicaSet with one pod in it. So it creates a one-replica deployment. Moving forward, let's go ahead and create a bare pod, or a pod that's not managed by a controller. And to create a bare pod, we use the command kubectl run, and then we're going to give it a name, hello-world-pod, and then we need to specify the container image that we want to run inside of that pod, and we do that again with the parameter, --image, and we're going to use our simple hello-world application again for this pod. We'll run that code, and we can see here at the bottom, pod/hello-world-pod created. And so let's check out the status of our deployment, and our bare pod, and I want to see if both of those pods are up and running, and I can do that with kubectl get pods. In the listing here, I have our pod that's associated with our deployment, that's the first one in the list there, and then our bare pod, which is hello-world-pod, which is the second one in the list there. We can see that both of those pods' status is running. Looking at the pod that's associated with our deployment, we can see in the name of the pod, the name of the deployment, which is hello-world, and then we see the string, 5457b44555. That's what's called the pod template hash, and is unique amongst ReplicaSets within a deployment. The last part of the pod name there is a unique identifier for a pod within a ReplicaSet, so gnxsk is a unique name. And so all of that together, with the deployment name, hello-world, the pod template hash, 5457b, and so on, plus the pod unique identifier, gnxsk, will give us the unique value for the pod name. So let's look at these pods from a different angle. Let's use kubectl get pods -o wide, to display some additional information about the pods that we have up and running, and I want to zoom in on IP and NODE. So we can see both of these pods have IPs that have been allocated from the pod network, 192.168.0.0/16. We can also see the nodes that these individual pods have been scheduled to. So the pod associated with our deployment, the first one there, was scheduled to c1-node3, and then our bare pod was scheduled to c1-node2. Now I want to point out that Kubernetes is a container orchestrator. It has the job of starting containers on nodes, and we can see that we have two pods up and running on two different nodes, and while those pods on those nodes started containers. And so what I want to do now is I want to SSH into an individual node, and show you how you can view the containers running on that node, even the ones that have been started by Kubernetes. And so we're going to go ahead and open an SSH connection into c1-node3, so I want to copy and paste this code from the top, down to the bottom, and SSH into c1-node3. That's the node where our pod that's

running from our deployment, was scheduled to and is now running on. The way that we can get a listing of the containers running on a node that are running with container D is to use the utility called crictl. And so I have the code to do just that on line 30 here. So we have sudo crictl, and then the parameter, --runtime-endpoint. And then we specify the cri socket for containerd, and then the command that we want to run there at the end, ps. And so if I highlight that code and run it, we can see that there are three containers running on this node in our cluster, the first one being our hello-app, which is part of our deployment. There are two other containers running on this node, one for our pod network, there we see calico-node, and we also see a container named kube-proxy, which is supporting our kube proxy pod, which is also running on this node. If you are still running Docker, here is the command to do just the same. If you aren't running containerd, you can do sudo docker ps, to get a listing of the containers that are running on your node. So let's go ahead and exit back out onto c1-cp1, and look at things from a couple other different angles. I want to show you some troubleshooting techniques that might be useful for working with your pods, first up, using kubectl logs. And so on line 41 here, I have the command kubectl logs hello-world-pod. We can use the command kubectl logs, which is very valuable to retrieve the logs from a container running inside of a pod in our cluster. And so any information written to standard out will be captured and available to you via kubectl logs. And so this is valuable when you have an application that's in trouble, or crashing, or a pod that won't start, so a very useful troubleshooting technique here. And so for our scenario, I have kubectl logs, and then a pod name. In this case, it's going to be hello-world-pod. So if I highlight that code, and run that, we can see the log from our container inside of our pod. In this case here, we have just one entry, which includes a date and timestamp, and then a string that says server listening on port 8080. Moving forward. I also want to show you that we can attach a shell to a running pod. And to do that, we can use a technique called kubectl exec. And I want to point out that you can use kubectl exec to start a process inside of a container, inside of a pod, and you can use this to launch any process, as long as that executable is available inside the container. In this scenario, we're going to attach a shell, so that I can show you how to have a shell to a container running inside of a pod. And so let's walk through that technique together. On line 47, I have kubectl exec -it. The parameter -it will allow you to attach an interactive terminal. We then specify the name of the pod that we want to attach it to. In this case, it's hello-world-pod. Then we have space minus minus, and then a space, which is a delimiter, and then after that, you specify the command that you want to run, in this case, /bin/sh, which is a shell. And so when I run that code, it's going to give me an interactive shell to the container running inside of that pod, and at the bottom here, you can see I have a root shell open. We can now execute any command that exists inside of the container. And so the first thing I want to show you is hostname. When I execute hostname, it'll print out the hostname of the pod stdout. And so here you can see the hostname, which by default will match the pod name, in this case, it's going to be hello-world-pod. If I do ip space addr, I can look at the network configuration of this individual container, running inside of this pod. Then we can see in the output, the IP of this pod is 192.168.131.62. When we're all done, we can use exit to exit out of the container, and get back onto our localhost, which is c1-cp1. Now, remember that first kubectl create deployment command that we executed in this demo? What that did is it created a deployment for us, which then created a ReplicaSet, which then created a pod, and I want to show you how that all pieces together here inside of our cluster. And so we'll first off do kubectl get deployment hello-world, and that's going to list the information about our deployment, hello-world. And so at the bottom here, we see the name, hello-world, we see that one of one pods is ready, and that our deployment was

created just under 9 minutes ago. If I do a kubectl get replicaset, we can see the ReplicaSet supporting the deployment. So there we see hello-world, and then the pod template hash, 5457b, and so on. We also can see the desired state and current state, and that one pod is up running, and ready, in our ReplicaSet. Now, if I do a kubectl get pods, we can see the pods that are up and running in the cluster, and now you can see where that name is built from, the deployment name, hello-world, then the pod template hash, and then that pod's unique identifier, which is gnxsk. So let's look a little more closely at the deployment, at its ReplicaSet, and its pods, and we'll use one of my favorite Kubernetes commands, kubectl describe, to do that. And so on line 64 here, I have kubectl describe deployment hello-world. I'm going to pipe that output into more, and here at the bottom we can see the the output of kubectl describe for our deployment. So we can see the name, hello-world, and the namespace default, and that it was created just a few minutes ago. Moving forward in this output, here we see pod template, which is the pod template for each of the pods that is created by this deployment. So inside of there, we can see containers, hello-app, and then the container image that we're working with, our hello-app from gcr. Going down to Events, we can see in Events that this deployment has an event that says ScalingReplicaSet, because the deployment creates the ReplicaSet, and then a ReplicaSet will go on and create the pods. So in the Events output at the bottom, we see ScalingReplicaSet from deployment-controller, Scaled up ReplicaSet hello-world-5457b44555 to 1. So that's creating that ReplicaSet, which will in turn create that one pod for us. Let's go ahead and look more closely at the ReplicaSet, and we can do that with kubectl describe replicaset hello-world. In this output here, we see the name, hello-world, and then the pod template hash. In Replicas, we can see the current state is 1, and desired is 1, and right below that, we see Pod Status one 1 is running. So we know currently we are in the desired state for this ReplicaSet. Moving forward, we see the pod template for this ReplicaSet. This matches the current state that is in our deployment. So there we see our containers, hello-app, running the container image from gcr, hello-app with the tag of 1.0. In the Events section for the ReplicaSet, we can see an event that created a pod, and so let's look at that output. We see SuccessfulCreate from replicaset-controller, Created pod: hello-world, and then our pod template hash, and then the pod identifier ending in gnxsk. Now let's go a level deeper and get kubectl describe information about our running pod. I want to show you a technique of how we can very quickly auto-complete a pod name at the command line with Bash auto-completion. So I'm going to take the highlighted text here on line 75, and thread it into my clipboard, and in the command line at the bottom, I'm going to paste that, and here we have kubectl describe pod hello-world and a dash. And so if I do a double tab now, it'll try to auto-complete to any pod that has the name that begins with hello-world-. And so we have two, right? We have the one associated with our deployment, and our bare pod, and so I'll just type a 5 there to give it enough information to auto-complete to the full pod name. By hitting tab again, we can see it auto-completes the full pod name for us. And so we have kubectl describe pod, and then our pod name, hello-world-5457b, and so on. I'm going to pipe that output into more, and here we can see the information associated with an individual pod. So there we have the pod name. We also can see the node that the pod is running on, so c1-node3, and its node IP address. Going down a little bit further, we can see the pod IP address. There we have 192.168.206.127. We can see that this pod is controlled by the ReplicaSet, hello-world-5457b, and so on, and then we can see the runtime information about the container, running inside of this pod. Going down a little bit further, I want to jump to the Events section, and in the Events section, we have some really good information about the lifecycle of this pod, and so let's walk through each one of these records. First up, we see Scheduled from the

default-scheduler, and the message is, Successfully assigned default, which is the namespace/, and then that's our pod name, hello-world-5457b, ending in gnxsk to c1-node3. So that's when the scheduling decision was made, and that's part of this pod's events. On the next line, we see Container image, and there's our container image, hello-app, with the tag of 1.0. We can see that it was already present on this machine, and so we didn't have to re-pull that down from the container registry; it already existed. So then we can jump right to creating the container image, and that's the third event in our output here, Created container hello-app. The last record we have in the Events is, Started container hello-app. And so that process of scheduling, pulling, creating, and starting, is all part of this pod's lifecycle. If you create a deployment and your pods don't come up, use kubectl describe on either the deployment, the ReplicaSet, or one of the pods in your deployment, and check out the events for those resources, looking for any events or errors that can help you understand what went wrong. This is my go-to place to help me troubleshoot deployment failures and pods failing to start up. Now, if you want to dive deeper into deployments, check out my course, Managing Kubernetes Controllers and Deployments, coming up later in this path. I have a link there for that. In that course, we'll talk about rollouts, controlling rollouts, rollbacks, and updating our application, so deep-dive stuff covered in that course.

Demo: Exposing and Accessing Services in Your Cluster

All right, so let's move forward into the next part of our demo where we're going to learn how to expose our deployment as a service and so we can access our application inside of our cluster. Now, to create a service for our deployment, we can do that with the code on line 89 here, kubectl expose deployment, and then the deployment name, hello-world \. The --port parameter is the port that the service is going to listen on. In this case, it's going to be on port 80. We then have a parameter target port, which is the port that our Pod is listening on. So we saw earlier that our application is listening on port 8080, so I'm going to highlight this code and run that to create the service for this particular deployment. And at the bottom, we can see that service hello-world is exposed. And so let's look at the information associated with that service, and we can do that with kubectl get service, specifying the service name hello-world. At the bottom here in the output, we can see the name of the service, the type is ClusterIP, and then the IP address associated with the service, or this is where we're going to send user traffic to is 10.100.236.189. There's no external IP associated with this. In a cloud scenario, you might see a public IP address there. The port that this service is listening on is port 80 on TCP. Let's look at this from another angle using kubectl describe, with kubectl describe service hello-world, and let's walk through this output together. We see a lot of the same information that we saw just a second ago, so we have the name, the type, and the IP address of the service. We see the port that we're listening on and also the target port. What I want to call out here is the Endpoints. Endpoints are the IP and port pairs for each of the Pods that are a member of the service. And currently there's only one Pod supporting this service, and so here we see the Pod IP and port 192.168.206.127 and port 8080. And so when traffic comes in on the service, it'll get routed to this endpoint. And so let's go ahead and do just that. Let's access the service inside of the cluster. And so what I'm going to do is I'm going to grab the IP address of the service and throw it into my clipboard and use curl to access our application, which is a simple Hello World web application, and curl is a command line web browser. If I type curl and then the protocol http://, and I'll paste in the service IP, and

then type in :80 for the port that we're listening on. When I press Enter, this is going to send traffic to the service IP, which will then get routed into the Pod that's supporting this particular service. So let's go ahead and press Enter and look at our output. Our output, simple Hello, World application running on version 1.0. The application also prints out the hostname. So here we can see the actual Pod name is the hostname. So we see hello-world, then our Pod template hash, and then the Pod ID ending in gnxsk. And so what happened there is we hit the service IP. The service then routes that traffic into the individual Pod supporting the service. If we scaled our application out, there would be additional endpoints registered into the service, and it would be up to kube-proxy to load balance that traffic amongst all of the Pods supporting the service. Now in the next part of the demo, what I want to show you here is how to use kubectl to generate YAML or JSON for resources that you have deployed in your cluster. In this case, we're going to look at a deployment. And so on line 116 here I have kubectl get deployment hello-world, and then the output modifier -o yaml, and I want to pipe that output into more. And so this will give us a YAML representation of the deployment object. This YAML representation also includes runtime information about the actual object itself. And so if we look inside of here, we see the YAML associated with the deployment, apiVersion apps/v1, kind is Deployment, but we also have a bunch of runtime information. This can be useful for monitoring in configuration management scenarios, but not so great as a source for manifest for declarative deployments. We'd have to remove all of this runtime information. I'm going to show you a technique in a few moments where we use dry run to help us generate these manifests very quickly and correctly at the command line without runtime information. If we also wanted to see this from a different angle, I can do the same, but for JSON. So kubectl get deployment hello-world, then with the output modifier -o json, pipe that into more. And here we can see the JSON representation of this object, including its runtime information. And so let's go ahead and clear out of that. And before we move forward into our next demo, I want to delete the resources that we created imperatively and recreate all of those declaratively in our upcoming demo. So let's walk through the process of deleting some resources together. Before we delete anything, let's look at what we have. And so with kubectl get all, we can see I have my deployment, which created my replicaset, which created the Pod supporting that deployment. We also see that our bare Pod is in there, and I also have two services up and running, one that we just created, our hello-world service, and a service for the API server that's available inside of the cluster. So let's delete our service, and we can do that with kubectl delete service hello-world. I'm going to go ahead and delete our deployment with kubectl delete deployment hello-world. What's going to happen when I do kubectl delete deployment hello-world, that'll delete the deployment, which will then delete the replicaset, which will then delete all the Pods associated with that replicaset. So there's a cascading delete that happens there. We then need to delete our bare Pod since it's not associated with any controller, and we can do that with kubectl delete pod, specifying a Pod name, hello-world-pod. That's going to block for a moment until that Pod is actually deleted, and then we'll get our console back. With that Pod deleted, if I do a kubectl get all now, all that we have left is that single cluster service for Kubernetes available.

Demo: Declarative Deployments and Accessing and Modifying Existing Resources in Your Cluster

Moving forward, let's look at how we can deploy resources declaratively in our cluster. We just walked through how to create things imperatively at the command line, but we want to get to where we're deploying things declaratively in code in our cluster, and so let's start that process together. On line 136 here, I have an example of where we can use a dry-run=client to help create a YAML manifest quickly and correctly, and so let's walk through the code to do just that. We have kubectl create deployment hello-world, and then specifying the image that we want to run. So really, that's no different than we saw in the previous demo. We then add on dry-run=client and -o yaml, and what that will do is create the YAML manifest for that deployment named hello-world running that container image. And that'll give us that output, the standard out. I'm going to pipe that into more. And so let's go and run this code and look at what we get. In the code at the bottom, we have the YAML that describes what we want to do. We want to create a deployment with that container image. So let's break out of this output and then run that code one more time, but instead of writing it to console, we'll write it to file. And so here, the only difference from the previous command is on line 144, where we're redirecting that output into a file named deployment.yaml file. And so from here we could take this deployment.yaml and build on a more complex deployment, if we need to, adding and changing the configuration inside of the deployment.yaml file if we needed to. And so let's take a peek at that just to make sure that we have what we want, and there we can see our code inside of there. For our demonstration here, we're going to take that file as is and deploy that in our cluster, and so let's look at the code to do just that. On line 152 we have kubectl apply -f deployment.yaml. And so what this will do is read that deployment.yaml file and send it into the API server for creation. So let's run that code and see what we get. A few moments later at the bottom we can see deployment.apps/hello-world is created. So we created our deployment declaratively in code. Now, we can do the same thing for our service. In a previous demo, we use the command kubectl expose deployment, and then specified the port as 80 and the target port as 8080. We can also add on dry-run=client -o yaml, and do the same thing, converting that imperative code into declarative code. And here we see at the bottom, we have our YAML representation of that. We'll take that code and run that into file, just like we did a few moments ago. But this time we're going to take this output for our service creation and redirect that into a file named service.yaml. We'll take a peek inside of there and look at the code inside of service.yaml, and here we can see the YAML manifest for the service that we want to create. I can't stress enough how valuable this technique is to quickly and correctly create manifests that you can either deploy right away or make simple modifications to that text before you send it into the API server for creation. So let's go ahead and create that service with kubectl apply -f service.yaml. At the bottom here we can see service/hello-world created. Let's check out the status of our deployment and our service, and we can do that with kubectl get all, run that code there, and here we can see all the resources deployed into our default namespace. In our default namespace we see our deployment hello-world. That created the replicaset here at the bottom. That replicaset created the Pod that we have up and running. We see our hello-world service and also our Kubernetes API service that are in the cluster. So next now, let's look at how we can make a change to an existing resource in our cluster. Since we have the code that describes our deployment, we can make a modification to that file and then just resubmit that into the API server and then make that change to our deployment. So let's do that together and edit deployment.yaml and make a modification to what we have

running in our cluster. So in here is the code that describes our deployment. If we go down into the spec, I want to jump over in replicas and go from 1 to 20. And so what I want to do is to change the number of replicas supporting our deployment from 1 to 20, and we're changing that in the code. But that doesn't change the application yet. I have to feed that code into the API server to effect that change. And we do that with kubectl apply -f, and we're going to send in deployment.yaml. At the bottom we can see deployment.apps/hello-world configured. We've effected that change to the desired state in our cluster. And now Kubernetes will go and spin up those 19 additional Pods to bring us up to 20 replicas supporting our deployment. And so if I do a kubectl get deployment now for hello-world, we can see in the output I have 20 of 20 Pods that are up, running, and ready. And if I do a kubectl get pods and pipe that into more, we can see the the whole collection of Pods that are up and running and supporting our deployment now. Now what happened behind the scenes when I scaled that deployment out, each one of those Pods that was a member of that deployment also got registered as an endpoint in the service and will automatically start receiving workload. And so if I do a kubectl get service now, we'll go ahead and grab the IP address of this new service, since I had to delete the previous service that we created imperatively and created a new one declaratively, the IP address changed. And so want to grab that IP address and throw that in my clipboard. We're going to use curl again, http://, and paste in the new IP address associated with our hello-world service. If I press Enter now, I'm going to access the service on the service IP and then get load balanced to 1 of the 20 Pods supporting this application. So there I got load balanced to a Pod ending in 7j8tc. Let's try this a few more times, and we can see each time that I access the pod, I get load balanced to a different Pod. So the next one we see is qvc6s, 5w5cg, we hit that one twice, and then we got load balanced to another Pod. And so kube-proxy is distributing that workload amongst 20 Pods that are a member of this service. Now let's say I didn't have that deployment manifest and I needed to scale my application or even make any other change to my application, and I don't have that deployment manifest. Well, I can use the command kubectl edit to edit a resource that's available in the cluster that's already up and running. And so let's look at the code to do this. Kubectl edit, what do I want to edit? I want to edit the deployment named hello-world. And so when I run this code, what it's going to do is retrieve the object via the API server and present that back to me in a local text editor. And so now I don't need to make a modification to the code and then send it in. What I'm going to do here is make the modification to this object in my text editor and then when I save this out, it'll send this code back into the API server and effect that change. And so here I'm going to change replicas from 20 to 30. And when I save this out, this is going to immediately be changed in our cluster effecting that change. So there we see deployment.apps/hello-world was edited. And so now the desired state of our cluster changed from 20 Pods to 30 Pods, and Kubernetes will go and scale up and create 10 more Pods in our cluster at the point in time in which that object was saved and sent back into the cluster when I exited my text editor. If I do a kubectl get deployment now for hello-world, we can see 30 of 30 Pods are up and running. I want to show you one other way to be able to scale an application at the command line, and we can do that with the command kubectl scale deployment. Which one? hello-world and specifying the --replicas parameter. Here we're going to set our number of replicas to 40, and at the bottom there we can see deployment.apps/hello-world is scaled. And if I check the status of the deployment with kubectl get deployment hello-world we can see 40 of 40 Pods are ready. Now I do want to call out that I continue to scale up the number of Pods here. We certainly could have reduced the number of Pods in any of these demonstrations to change the desired state of our application. I'll leave that as an exercise to you, the viewer, to

experiment with that. And so that's a wrap for this demo. Let's go ahead and clean up our resources that we deployed, and I'm going to use kubectl delete deployment hello-world to delete that and then kubectl delete service hello-world to delete our service. And then I'll do a kubectl get all, and here we can see we're kind of in an intermediate state, where those Pods are all getting terminated actively for us behind the scenes. And so we still have a couple that are up and running, and a few moments later, all of these Pods will be shut down and deleted from the cluster, and all that we'll be left with Is that Kubernetes service running in our cluster. Let's refresh kubectl get all one last time to see, and all that we have left is our Kubernetes cluster API service.

Module Summary and Thank You!

Here we are at the end of our module, and we introduced how we can interact with our cluster focusing on using kubectl to retrieve different types of information out of our cluster. And then we looked closely at how to deploy some applications and what is really happening behind the scenes. We discussed both imperative and declarative deployment methods, and hopefully I made the case for using declarative deployment methods. So here we are at the end of our course, and I really hope you enjoyed listening to this and that we laid the appropriate foundation for your Kubernetes studies. We covered a lot of ground together so far. We discussed the Kubernetes architecture, we did a deep dive into installing a Kubernetes cluster, both on-prem and in the cloud, and we looked at how to interact with our cluster using kubectl and did some application deployments. It's truly been a pleasure recording this course, and I thank you so much for listening, and most importantly, learning with me. I hope you enjoyed the course. Join me again soon, here at Pluralsight.