Learn Kubernetes TV Script

Table of Contents

[V001: 2](#_Toc178630469)

[V002: 5](#_Toc178630470)

[V003 7](#_Toc178630471)

[V004 11](#_Toc178630472)

[V005 14](#_Toc178630473)

# V001:

**[Opening]**

Hello and welcome to learn Kubernetes TV!

Have you ever wanted to learn Kubernetes in a production environment like AWS but didn’t know where to start? Today, I’m going to show you how easy it is with a completely hands-on approach.

In this series we will go through all different building blocks of Kubernetes and learn how everything works.

By the end of the first video, you’ll have your very own Kubernetes cluster running on AWS using simple tools like AWS CLI, eksctl, and kubectl.

**[Prerequisites Setup]**

Before we dive in, there are three tools you'll need to install on your machine. First up is the AWS CLI, which allows us to interact with AWS services from our command line. You can find the installation guide on the AWS website.  
[*https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html*](https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html)

Next, we’ll need eksctl, a CLI tool that simplifies Kubernetes cluster management on AWS. You can follow this link to install it.  
[*https://eksctl.io/installation/*](https://eksctl.io/installation/)

Finally, we need kubectl, the command-line tool for Kubernetes. Here’s the link to install it based on your OS.  
[*https://kubernetes.io/docs/tasks/tools/install-kubectl-windows/*](https://kubernetes.io/docs/tasks/tools/install-kubectl-windows/)

**[AWS Credentials Setup]**

Once we’ve installed the necessary tools, let’s configure our AWS credentials. For this, you'll need an IAM user with the appropriate permissions to manage EKS clusters.

Show the IAM console and the credentials.

Once your IAM user is set up, AWS credentials are stored in the .aws/credentials file. Here's how to view that file on Windows.

cat ~/.aws/credentials

**[Check AWS Account and Credentials]**

Next, let's confirm that everything is set up correctly by checking our AWS account and listing S3 buckets. This is a quick way to ensure your credentials are correctly configured.

aws sts get-caller-identity

aws s3 ls

**[Creating a Kubernetes Cluster with eksctl]**

Alright, it’s time to create our Kubernetes cluster using eksctl. This command will set up a cluster in the eu-west-1 region, using Fargate for serverless pods, and tagging it as a dev environment.

eksctl create cluster \

--name lkt-temp-04 \

--region eu-west-1 \

--fargate \

--version 1.30 \

--tags "env=dev,owner=engineering"

While the cluster is being created, let’s take a quick look at some helpful eksctl commands.

eksctl --help

eksctl info

eksctl get cluster

**[Configuring Kubernetes Access]**

Once the cluster is ready, we need to configure access locally. We do this by updating our kubeconfig file, which is used by kubectl to interact with Kubernetes.

aws eks --region eu-west-1 update-kubeconfig --name lkt-temp-03

This will save the cluster's credentials to the ~/.kube/config file. If you want, you can also set this config path as an environment variable using $KUBECONFIG.

A Kubernetes context is a set of configuration data used by kubectl to access a specific cluster. It includes details like the cluster’s API server URL, your user credentials, and the namespace you're working in. By switching contexts, you can easily manage multiple Kubernetes clusters from one command-line setup.

**[Using kubectl to Manage the Cluster]**

Now that we have access to the cluster, let’s use kubectl to check the current context, which defines which cluster kubectl is interacting with.

kubectl config current-context

You can list all available contexts with this command.

kubectl config get-contexts

You can view the content of your config file with this command.

kubectl config view

And if you need to switch between contexts, here’s how you do it.

kubectl config use-context arn:aws:eks:eu-west-1:047838238778:cluster/lkt-temp-03

"Finally, if you no longer need a context, you can delete it like this."

kubectl config delete-context arn:aws:eks:eu-west-1:047838238778:cluster/lkt-temp-03

**[Viewing Kubernetes Pods]**

To wrap up, let’s take a look at all the pods running in our cluster. Use this command to list all pods in all namespaces.

kubectl get pod -A

**[Conclusion]**

That’s it! You now have a Kubernetes cluster up and running on AWS.

In the next video we will look into the anatomy of each object and investigate how we can create new object.

If you have any questions or run into any issues, feel free to drop them in the comments below. And if you found this video helpful, don’t forget to like, subscribe, and hit the bell icon for more hands-on Kubernetes tutorials. Thanks for watching, and see you soon on Learn Kubernetes TV!

# V002:

**[Introduction]**

Hey everyone, welcome back to *Learn Kubernetes TV*!

In the last video, we built a basic Kubernetes cluster. If you missed it, be sure to check that out for a solid foundation.

Today, we’re going to dive deeper into understanding your Kubernetes cluster, the API resources available to you, and some essential kubectl commands to manage and troubleshoot resources effectively. So let's get started!"

**[Get Cluster Information]**

We’ll begin by checking the cluster we're connected to with kubectl cluster-info. This gives us an overview of the essential services running in the cluster like the Kubernetes API server and the DNS service.

kubectl cluster-info

Here you’ll see the URLs for the Kubernetes API server and other components. It’s a quick way to ensure you're connected to the right cluster.

**[Listing Available API Resources]**

Next, let’s explore all the API resources available in the cluster using the kubectl api-resources command.

kubectl api-resources

This command lists all the resources you can interact with—like Pods, Deployments, Services, ConfigMaps, and more. Each resource type has its unique properties and functionality, which we’ll be covering throughout the series.

Talk about resource names, short names, namespace and kind.

**[Exploring Resource Structure with kubectl explain]**

Now, let’s dive deeper into how these resources are structured using kubectl explain. For example, let's take a look at Pods.

kubectl explain pods

kubectl explain pods.spec

kubectl explain pods.spec.containers

It provides detailed documentation about Kubernetes resources. It helps users understand the various fields of a resource and their purpose. This can be particularly useful for those who are new to Kubernetes or for experienced users who need a quick reference.

**[kubectl apply vs kubectl create]**

Now, a really important concept to understand is the difference between kubectl apply and kubectl create. Both are used to create resources, but they work very differently.

* **kubectl apply is idempotent**, meaning you can run it multiple times, and it will only apply changes if necessary. It can both create and update resources.
* **kubectl create is not idempotent**—it will fail if the resource already exists, and it’s only meant for creating new resources, not updating existing ones.

"Let’s see this in action."

kubectl apply -f pod1.yaml

kubectl get pod

kubectl delete pod nginx

kubectl create -f pod1.yaml

kubectl get pod

Notice how kubectl apply can handle updates and creation. Always prefer using kubectl apply when you're managing resources over time.

**[Dry-Run Modes for Safer Changes]**

One of the best practices in Kubernetes is to use dry-run modes to test changes before applying them. There are two types of dry-run options:"

* **--dry-run=client**: This checks the YAML manifest for syntax errors without connecting to the cluster.
* **--dry-run=server**: This checks if the manifest will be accepted by the Kubernetes API server, ensuring that resource types and configurations are valid without making any actual changes.

"Let’s test this with some examples."

kubectl apply -f deployment.yaml --dry-run=client

kubectl apply -f deployment-error1.yaml --dry-run=client

kubectl apply -f deployment-error1.yaml --dry-run=server

kubectl apply -f deployment-error2.yaml --dry-run=client

kubectl apply -f deployment-error2.yaml --dry-run=server

"By using --dry-run, you can catch potential issues early without impacting the cluster."

**[Generating YAML Manifests]**

Sometimes you’ll want to generate a resource’s YAML manifest without actually applying it, so you can tweak it or save it for later use. Let’s generate a Deployment YAML for an NGINX application.

kubectl create deployment nginx --image=nginx --dry-run=client -o yaml

kubectl create deployment nginx --image=nginx --dry-run=client -o yaml > deployment-dry-run.yaml

kubectl apply -f deployment-dry-run.yaml

kubectl get deployment

kubectl delete deployment nginx

"This method gives you flexibility to generate and customize your manifests before applying them."

**[Using kubectl diff]**

"Another powerful tool is kubectl diff, which allows you to compare your current resources against what’s defined in your YAML file. This is useful for identifying what changes will be applied before running kubectl apply.

kubectl apply -f deployment.yaml

kubectl diff -f deployment-modified.yaml

kubectl apply -f deployment-modified.yaml

kubectl get deployment

kubectl delete deployment hello-world

kubectl diff shows you a side-by-side comparison of what will change. It's especially handy when you're working in production environments where you want to be cautious about changes.

**Conclusion**

"That’s a wrap for this session! Today we covered some powerful kubectl commands, learned how to explore resource structures with kubectl explain, and discussed dry-run modes and how to generate YAML files for future use. I hope you're feeling more confident navigating Kubernetes.

In the next video, we’ll dive even deeper into Kubernetes operations on AWS, so stay tuned! Don’t forget to like, share, and subscribe to *Learn Kubernetes TV* for more hands-on tutorials. Thanks for watching and see you in the next video!

# V003

**[Introduction]**

Hey everyone, welcome back to *Learn Kubernetes TV*!

Today, we’re going to explore Kubernetes API resources in more depth, understand how API requests work, and dive into different ways to interact with your cluster. We’ll also cover logging, debugging, and use kubectl proxy to access API resources. Let's jump in!"

**[Exploring API Resources]**

Let’s start by listing all API resources available in our Kubernetes cluster.

kubectl api-resources

kubectl get deployment --all-namespaces

kubectl get deploy -A

kubectl get pod --all-namespaces

kubectl get po -A

kubectl api-resources gives us a list of all resource types in our cluster.

With kubectl get, we can list a resource type like deployments and pods in specific namespace or across all namespaces using the -A flag.

The shorthand po refers to pods, and deploy refers to deployments, just to save some typing.

**[API Groups]**

API groups are to organize Kubernetes API resources for better management and versioning.

Core Group: Essential resources like Pods and Services (e.g., v1).

Extended Groups: Other groups like apps (Deployments), batch (Jobs), and networking.k8s.io (NetworkPolicies).

Each resource belongs to an API group, and we can filter resources based on the API group we’re interested in. For example, to get the resources under the apps group or the rbac.authorization.k8s.io group, we can use the following commands.

kubectl api-resources --api-group=apps

kubectl api-resources --api-group=rbac.authorization.k8s.io

API groups help us organize resources in Kubernetes by functionality.

**[Explaining Resources by API Version]**

Next, we’ll use kubectl explain to view the structure of a resource. Here, I’ll show you how to view the structure of a pod with a specific API version.

kubectl explain pod --api-version v1

This is super helpful for understanding the exact structure of resources and what fields are available for different API versions.

**[Understanding an API Request]**

Let’s now apply a YAML manifest and see how Kubernetes handles the API request.

kubectl apply -f pod2.yaml

kubectl get pod hello-world

This applies a pod configuration and then retrieves the pod named hello-world. In this case, Kubernetes will handle the resource creation request based on the structure in the YAML file.

**[Output Verbosity and Debugging Levels]**

Sometimes you need more detail to debug what’s happening under the hood. kubectl supports output verbosity levels, ranging from 0 to 9. Let’s see what additional details we get at different verbosity levels.

kubectl get pod hello-world -v 6

kubectl get pod hello-world -v 7

kubectl get pod hello-world -v 8

kubectl get pod hello-world -v 9

As you increase the verbosity level, more information is shown about the underlying API requests and responses. Higher levels can be very useful when troubleshooting.

**[Using kubectl proxy to Authenticate API Requests]**

Next, we’ll use kubectl proxy to set up a local proxy that allows us to authenticate and send API requests directly to the API server. This is especially useful for accessing Kubernetes resources without needing direct access to the API server itself.

kubectl proxy --port=8080 &

curl http://localhost:8080/api/v1/namespaces/default/pods/hello-world

This starts a proxy on port 8080, and with the curl command, we can directly query the API server for information about the hello-world pod.

**[Getting Pod Details in Different Ways]**

Now let’s explore different ways to retrieve detailed information about our hello-world pod.

kubectl get pod hello-world

kubectl describe pod hello-world

kubectl get pod hello-world -o yaml

kubectl get gives a basic overview, kubectl describe gives us detailed information including events, and kubectl get -o yaml gives us the raw YAML manifest of the pod.

**[Monitoring Pods with the --watch Flag]**

One of the cool features of kubectl is the --watch flag. It continuously monitors changes to the resources you’re watching.

kubectl get pods --watch -v 6

You can see real-time updates of pod status in your terminal. Now, let’s delete the pod and watch how kubectl tracks that event.

kubectl delete pods hello-world

**[Accessing Pod Logs]**

Let’s bring back the hello-world pod by applying the manifest again.

kubectl apply -f pod2.yaml

Logs are essential for debugging. Let’s access the logs of the hello-world pod.

kubectl logs hello-world

kubectl logs hello-world -v 6

We can also retrieve logs via the API server by using kubectl proxy again.

kubectl proxy --port=8080 &

curl http://localhost:8080/api/v1/namespaces/default/pods/hello-world/log

**[Simulating Authentication Failures]**

Now, let’s simulate an authentication failure by tampering with our kubeconfig file. First, we’ll back it up.

cp ~/.kube/config ~/.kube/config.main

Next, we edit the username in the kubeconfig file.

vi ~/.kube/config

kubectl get pods -v 6

As expected, this results in an authentication error.

**[Restoring Kubeconfig]**

Now let’s restore our kubeconfig file and get the pods again.

cp ~/.kube/config.main ~/.kube/config

kubectl get pods

We’re back to normal!

**[Handling Missing Resources]**

What happens when you try to get a resource that doesn’t exist? Let’s see how Kubernetes responds.

kubectl get pods nginx-pod -v 6

As you can see, we get a 404 error because the resource was not found.

**[Creating and Deleting a Deployment]**

Finally, let’s create a deployment, check its status, and then delete it. Kubernetes returns a 404 for not finding the deployment and a 201 for creating it.

kubectl apply -f deployment.yaml -v 6

kubectl get deployment

Now, let’s delete the deployment.

kubectl delete deployment hello-world -v 6

As you can see, Kubernetes returns a 200 OK for both the DELETE and GET requests, confirming that the operation was successful.

**Conclusion**

That wraps up today’s video! We’ve covered many topics about Kubernetes API, including how API requests work, verbosity levels, and more. You’ve learned how to use kubectl proxy, access pod logs and gather information about your pods. In the next video, we’ll dive even deeper into managing namespace.

If you enjoyed this content, please like, share, and subscribe to Learn Kubernetes TV so you don’t miss out on future videos. Thanks for watching, and see you next time!

# V004

**[INTRO]**

Welcome back to Learn Kubernetes TV! In today’s video, we’re diving into namespaces, a key concept in Kubernetes, and exploring practical commands to help you manage them. We’ll also cover how to create deployments and interact with resources across namespaces.

Think of namespaces like Windows folders that help organize your resources. In Kubernetes, these resources include pods, deployments, config maps, and more.

**[Listing Namespaces and API Resources]**

Let’s kick things off by listing all the namespaces in our Kubernetes cluster.

kubectl get namespaces

This shows the different environments or 'spaces' in which your Kubernetes objects are running.

Next, let’s explore the API resources to see which ones can exist in a namespace and which can’t.

kubectl api-resources --namespaced=true | head

kubectl api-resources --namespaced=false | head

Here, we are seeing resources that live inside namespaces and those that exist cluster-wide, like nodes and persistent volumes.

**[Exploring Namespace Details]**

To dive deeper, let's describe the status of all namespaces.

kubectl describe namespaces

And if we want to focus on a specific namespace like kube-system, we can do this:

kubectl describe namespaces kube-system

This gives you an in-depth look at the details of each namespace, such as their labels, annotations, and status.

**[Resource Quotas and Limits]**

Kubernetes namespaces support **resource quotas** and **limit ranges** to control resource usage.

* **Resource Quotas** set the maximum amount of resources (CPU, memory, storage) a namespace can consume, ensuring fair resource allocation across teams or projects.
* **Limit Ranges** define default and maximum resource requests and limits for pods or containers, preventing any single container from over-consuming resources and impacting other workloads.

These tools help manage resources effectively within a shared cluster.

kubectl apply -f ns-limit.yaml

kubectl apply -f ns-quota.yaml

kubectl describe limitrange -n kube-system

kubectl describe resourcequota -n kube-system

kubectl delete limitrange example-limits -n kube-system

kubectl delete resourcequota example-quota -n kube-system

**[Working with Pods Across Namespaces]**

Now, let’s list all the pods across all namespaces.

kubectl get pods --all-namespaces

kubectl get pods -A

This is useful when you want to see everything running in your cluster, regardless of namespace. If you want to filter pods only in the kube-system namespace:

kubectl get pods --namespace kube-system

**[Creating and Deleting Namespaces]**

Namespaces are great for organizing resources. Let’s imperatively create a new namespace called pandora1.

kubectl create namespace pandora1

**[Tip]** "Remember, Kubernetes namespaces must be in lowercase and can include dashes. If we try to create Pandora1, Kubernetes will reject it."

kubectl create namespace Pandora1

Now, let's create a namespace declaratively using a YAML file.

kubectl apply -f namespace.yaml

"And finally, to see all namespaces again:"

kubectl get namespaces

**[Deploying into Namespaces]**

Next, let’s deploy an application into our pandora1 namespace.

kubectl apply -f deployment.yaml

This deployment creates 4 replicas of the hello-world app in the pandora1 namespace.

In addition to creating resources declaratively with YAML, Kubernetes also allows you to create resources imperatively. Let’s create a simple pod using the kubectl run command.

kubectl run hello-world-pod \

--image=gcr.io/google-samples/hello-app:1.0 \

--namespace pandora1

Here, we are creating a pod called hello-world-pod running the hello-app container from Google’s sample image in the pandora1 namespace. This is a quick way to create resources without needing to write a YAML file.

However, neither the deployment nor the pod was created successfully. This is due to the default Fargate profile's configuration.

kubectl get pods --namespace pandora1

**[Creating Fargate Profiles and Cleaning Up]**

"Now, let’s create a Fargate profile for our pandora1 namespace using eksctl."

eksctl create fargateprofile \

--cluster lkt-temp-03 \

--name fp-pandora \

--namespace pandora1inyaml \

--namespace pandora1

Fargate profiles allow us to run pods in AWS Fargate, which is a serverless compute engine for containers.

Next, we’ll delete all existing pods and recreate them. After the new pod is created, we can check its status:

kubectl get pods --namespace pandora1

And if we want to see only the pods in the pandora1 namespace:

kubectl get pods --namespace pandora1

This command verifies that our pod has been successfully created in the correct namespace.

**[Managing Pods and Resources]**

We can now check if the pods are running in the pandora1 namespace.

kubectl get pods --namespace pandora1

kubectl get pods -n pandora1

kubectl get pods -A

All these commands will list the pods either in a specific namespace or across all namespaces.

To list **all** resources in the pandora1 namespace, use:

kubectl get all --namespace=pandora1

Finally, let’s clean up by deleting all the pods in our pandora1 namespace. Keep in mind that pods managed by a deployment will be recreated automatically.

kubectl delete pods --all --namespace pandora1

kubectl get pods -n pandora1

And if you’re done with the namespace entirely, you can delete it along with all its resources.

kubectl delete namespaces pandora1

kubectl delete namespaces pandorainyaml

**[OUTRO]**

That wraps up today’s session on managing namespaces, deploying applications, and working with Fargate profiles in Kubernetes. In the next video we will talk about labels, selector and scheduling pods.

If you enjoyed this content, please like, share, and subscribe to Learn Kubernetes TV so you don’t miss out on future videos. Thanks for watching, and see you next time!

# V005