

++Setting sail on your Kubernetes voyage

Kubernetes has become the standard for deploying and managing distributed applications, used by companies like Spotify, Pinterest, and CERN. While complex at first glance, this course aims to simplify its learning curve. By the end, learners will understand why Kubernetes is so widely adopted, grasp its core architecture, and learn best practices for security and troubleshooting. Led by Kim Schlesinger, this course offers real-world Kubernetes experience for anyone eager to learn container orchestration.

++What you should know

To begin this course, you need a basic setup: a computer with terminal access (Windows/Linux/macOS), internet connectivity, and a text editor like VS Code. Prior knowledge of container tech (e.g., Docker or Podman) is helpful, even at a surface level. Familiarity with basic Unix commands (`mkdir`, `cd`) is required. Kubernetes manifests (YAML files) will be edited, so a foundational understanding of containers and interest in orchestration systems is enough to start.

++What is Kubernetes?

Kubernetes (also known as K8s or Kates), launched by Google in 2014 and donated to CNCF in 2015, is an open-source container orchestration system written in Go. Inspired by Google's internal system Borg, Kubernetes automates deployment, scaling, and management of containerized applications. It provides autoscaling, failover, and load balancing capabilities across on-prem, cloud, or hybrid environments. Major companies like Spotify rely on it for handling millions of requests per second due to its elasticity and automation. Kubernetes supports both manual installation and managed services from cloud providers like AWS, GCP, Azure, and DigitalOcean, making it scalable from startups to enterprises.

++What are containers?

Containers, popularized by Docker, are lightweight, portable units that package application code along with everything needed to run it—runtime, system tools, libraries, and dependencies. Unlike virtual machines, containers are more efficient as they share the host OS kernel and require less CPU and memory, allowing more containers per server. They are highly scalable and can spin up or down in seconds. A *container image* is a static file with executable content, while a *container registry* (like Docker Hub or Google Container Registry) is a repository for these images. Kubernetes uses a container runtime (e.g., Docker, containerd, Podman) to fetch images and launch containers as specified. In this course, pre-built images are used, and Kubernetes will manage replication and scaling. The nautical theme—Kubernetes as the "ship captain" and Docker as the "container ship"—adds to the metaphorical appeal.

++What is cloud native?

Cloud native refers to designing software to run in scalable, automated environments like public, private, or hybrid clouds. The Cloud Native Computing Foundation (CNCF), part of the

Linux Foundation, defines cloud native technologies as open-source solutions that enable automatic deployment and scaling using tools like containers, microservices, service meshes, immutable infrastructure, and declarative APIs. Unlike the slow, siloed development workflows of the past, cloud native practices break down barriers between dev, ops, and infrastructure teams to enable continuous delivery. CNCF supports projects by categorizing them into sandbox (experimental), incubating, and graduated (production-ready). Kubernetes, its first graduated project, benefits from CNCF support through community, documentation, certifications, and global events like KubeCon. Other CNCF projects include Helm (package manager), Prometheus (monitoring), and Linkerd (service mesh).

++Install Docker on Windows

To run a local Kubernetes cluster, Docker Desktop must be installed first. Docker acts as the container engine that runs your containers locally. On Windows 10, the installation process involves downloading Docker Desktop from the official Docker Docs site, executing the installer, enabling shortcuts, and restarting the computer after installation. Once Docker Desktop is set up, tools like Minikube can be installed to simulate a Kubernetes cluster locally. This enables you to deploy and manage Kubernetes resources in a controlled, developer-friendly environment.

Reading and Writing YAML

YAML is used to define Kubernetes resources in a human-readable format. Understanding indentation, key-value pairs, and basic syntax is essential. You write configurations for Pods, Services, Deployments, etc., using YAML files, which Kubernetes then uses to create and manage resources.

Namespaces allow you to divide cluster resources between users or projects. You can create a namespace using the command `kubectl create namespace <name>`, helping organize workloads, apply RBAC rules, and avoid name collisions.

Deploy an Application

You can deploy an application by creating a YAML file (e.g., `deployment.yaml`) and applying it using `kubectl apply -f <file>`. This file typically defines metadata, spec, and replicas of the application using a Deployment resource.

✓ Check the Health of a Pod Using Event Logs

Use `kubectl describe pod <pod-name>` to see event logs and check pod health. Events show issues like image pull errors, failed mounts, or crash loops, which are critical for debugging.

✅ Check That Your Application is Working with BusyBox

BusyBox is a lightweight Linux tool often used for debugging. You can run it inside a pod and use it to ping services or perform curl commands, ensuring connectivity and application responsiveness within the cluster.

✅ View Your Application Logs

Use `kubectl logs <pod-name>` to fetch logs for a pod. This helps track the application's behavior and debug runtime errors by reviewing output printed by the app inside the container.

✅ Install Minikube on Windows

Minikube allows you to run a local Kubernetes cluster. After installing Docker, you download and run the Minikube installer on Windows. It sets up a single-node cluster for testing and learning Kubernetes without needing cloud infrastructure.

- Only **ClusterIP**, **NodePort**, **LoadBalancer**, and **ExternalName** are built-in service types.

Section 5: Kubernetes Architecture

🧩 The Kubernetes Control Plane

- Manages the entire cluster; it's the **brain** of Kubernetes.
- Core components:
 - **kube-apiserver**: The front door; exposes the Kubernetes API (RESTful interface).
 - **etcd**: Consistent and distributed key-value store; holds all cluster data.
 - **kube-scheduler**: Assigns pods to suitable nodes based on resource availability, affinity rules, etc.

- **kube-controller-manager**: Runs controllers (e.g., ReplicationController, NodeController).
- **cloud-controller-manager**: Integrates Kubernetes with cloud provider APIs (optional).

🧠 *Sample Q: What happens if etcd is corrupted?*

→ All cluster state is lost unless backed up. It's critical to secure and regularly back up etcd.

🔧 Kubernetes Worker Nodes

- Where **containers actually run** (inside pods).
- Each node runs:
 - **kubelet**: Talks to the API server, ensures the node runs assigned pods.
 - **kube-proxy**: Manages networking rules and service routing on nodes.
 - **Container runtime**: e.g., Docker, containerd – pulls and runs container images.

🧠 *Sample Q: What happens if kubelet crashes on a node?*

→ The node can't report its status or manage pods. It may be marked as NotReady.

🤝 How the Control Plane and Nodes Work Together

- Control plane instructs nodes via the API.
- Nodes respond with status updates via kubelet.
- Communication is bidirectional and secured via TLS.
- **Controller** loops constantly reconcile the desired state with the actual state.

🧠 *Sample Q: What if a pod is deleted manually from a node?*

→ The controller recreates it — Kubernetes ensures the declared state matches reality.

Kubernetes Cluster Components Summary

- Control Plane: API server, etcd, scheduler, controllers.
- Node Components: kubelet, kube-proxy, container runtime.
- Others: DNS, dashboard, Ingress controllers, CSI drivers.

 *Sample Q: Is kube-proxy needed on all nodes?*

→ Yes, for handling internal/external traffic to services.

Section 6: Advanced Topics

Ways to Manage Kubernetes Pods

- You don't create pods directly in production — instead use:
 - **Deployments** (for stateless apps)
 - **StatefulSets** (for stateful apps)
 - **DaemonSets** (1 pod per node, e.g., for logging agents)
 - **Jobs/CronJobs** (for batch and scheduled tasks)

 *Sample Q: When would you use a DaemonSet?*

→ For tasks that must run on every node, like a metrics collector.

Running Stateful Workloads

- Use **StatefulSets** to manage pods with:
 - Stable, unique network identity (**pod-0**, **pod-1**)
 - Persistent volumes that stick with the pod
- Works well for DBs like Cassandra, MongoDB, etc.

🧠 *Sample Q: How does a StatefulSet maintain identity?*
→ By using persistent volume claims (PVCs) and stable DNS naming.

Kubernetes Security

- Key principles:
 - **RBAC** (Role-Based Access Control): Fine-grained permission system.
 - **Namespaces**: Isolate resources between teams/environments.
 - **Network Policies**: Define what pod can talk to what.
 - **Pod Security Standards**: Prevent privilege escalation, restrict container capabilities.
 - **Secrets & ConfigMaps**: Store sensitive data/config separately from code.

🧠 *Sample Q: Why not store secrets in environment variables?*
→ Not encrypted; better to use Kubernetes Secrets, which can be mounted as volumes or env vars securely.

yes

ChatGPT said:

Awesome! Here's a set of **mock Kubernetes quiz/viva questions and answers** based on the sections you've covered. These go from moderate to tough and include quick explanations to help with understanding and recall.

Section 5: Kubernetes Architecture

1. Q: What is the role of the `kube-apiserver` in the control plane?

A: It acts as the front-end for the Kubernetes control plane and exposes the Kubernetes API. All other components interact with the API server.

2. Q: If the `etcd` store is corrupted, what happens?

A: The cluster's state may be lost, as etcd holds all the configuration and status data. Regular backups are crucial.

3. Q: How does the `kube-scheduler` decide where to place a pod?

A: It considers available resources, affinity/anti-affinity rules, taints/tolerations, and other constraints to pick a suitable node.

4. Q: What's the difference between a `controller` and a `scheduler` in Kubernetes?

A: The scheduler decides *where* a pod goes; controllers (like ReplicaSet) ensure *what should exist* does exist by watching and reacting.

5. Q: What components must exist on every Kubernetes worker node?

A: `kubelet`, `kube-proxy`, and a container runtime (like containerd or Docker).

6. Q: What does the `kube-proxy` do?

A: It handles network routing, exposing services and managing NAT rules so services can be reached across the cluster.

7. Q: Can you run a Kubernetes cluster without a container runtime?

A: No, container runtimes are essential to run containers inside pods. Examples: Docker, containerd, CRI-O.

Section 6: Advanced Topics

8. Q: What is a DaemonSet and when should you use it?

A: A DaemonSet ensures a pod runs on every node. Useful for logging agents, monitoring tools, or any node-specific task.

9. Q: What is the main difference between a Deployment and a StatefulSet?

A: Deployments manage **stateless** pods with interchangeable identities, while StatefulSets manage **stateful** pods with stable identities and persistent storage.

10. Q: What Kubernetes resource would you use to schedule a recurring backup job?

A: A **CronJob**.

11. Q: How does Kubernetes manage sensitive information like database credentials?

A: Using **Secrets**, which can be mounted into pods as environment variables or files and are base64-encoded.

12. Q: What is RBAC in Kubernetes?

A: **Role-Based Access Control** is a method of defining what users or service accounts can do (verbs) on which resources in the cluster.

13. Q: What is a NetworkPolicy and what does it control?

A: A NetworkPolicy defines how pods can communicate with each other and with outside networks. It restricts traffic based on labels and selectors.

14. Q: Name one pod security policy best practice.

A: Don't run containers as root. Use **runAsUser**, drop **CAP_SYS_ADMIN**, and prevent privilege escalation with **allowPrivilegeEscalation: false**.

