From Docker Compose to Kubernetes

Part 1: Kubernetes Fundamentals

A hands-on workshop for developers

Transitioning from docker-compose.yml to Kubernetes

Workshop Overview

Part 1: Fundamentals (4 hours)

- Environment Setup
- Core Resources: Pods, Deployments, Services
- Configuration: ConfigMaps & Secrets
- Storage: Volumes & Persistence
- Organization: Namespaces & Management
- Final Integration Lab

Part 2: Advanced Topics (4 hours)

- Ingress, Helm, GitOps
- Monitoring, Security, Autoscaling

Prerequisites

Required Knowledge:

- Docker and docker-compose experience
- Basic Linux command line
- YAML syntax
- Container concepts

What You'll Learn:

- Kubernetes core concepts
- Migrating from Compose to K8s
- kubectl and k9s tools
- Best practices for K8s manifests

Environment Setup

Tools Used:

- Podman (container runtime)
- kind (Kubernetes in Docker)
- kubectl (K8s CLI)
- k9s (Terminal UI)
- Workshop container with pre-installed tools

Cluster Setup:

```
$ kind create cluster --name workshop --config=kind-simple.yaml
$ kubectl cluster-info
```

Why Kubernetes?

Docker Compose:

- Simple, single-host deployments
- Great for dev environments
- Limited scaling and orchestration

Kubernetes:

- Multi-node, production-ready
- Self-healing and auto-scaling
- Advanced networking and storage
- Declarative configuration
- Industry standard for container orchestration

Kubernetes Architecture

Control Plane:

- API Server (entry point)
- Scheduler (pod placement)
- Controller Manager (desired state)
- etcd (cluster data store)

Worker Nodes:

- kubelet (pod management)
- kube-proxy (networking)
- Container runtime

Part 1.1: Introduction to Kubernetes

Key Concepts:

- Declarative vs. Imperative
- Desired state reconciliation
- Labels and Selectors
- Namespaces

First Command:

```
$ kubectl get nodes
$ kubectl cluster-info
```

Part 1.2: Environment Setup

Workshop Container:

```
$ podman run -it --name kind-workshop \
  --privileged \
  -v ${PWD}:/workspace \
  fedora:39 bash
```

Cluster Types:

- Simple: 1 control-plane + 1 worker
- Multi-node: 1 control-plane + 3 workers
- HA: 3 control-planes + 3 workers

Part 1.3: Pods - The Smallest Unit

What is a Pod?

- One or more containers
- Shared network namespace
- Shared storage volumes
- Ephemeral by nature

Docker Compose Service:

```
services:
  app:
    image: nginx:1.25-alpine
```

Pods - Kubernetes Equivalent

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: app
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.25-alpine
    ports:
    - containerPort: 80
```

Create Pod:

```
$ kubectl apply -f pod.yaml
$ kubectl get pods
```

Multi-Container Pods

Sidecar Pattern:

```
spec:
  containers:
  - name: app
    image: myapp:latest
  - name: logger
    image: fluentd:latest
```

Use Cases:

- Logging sidecars
- Proxy/service mesh
- Initialization helpers

Part 1.4: Deployments

Why Deployments?

- Manage ReplicaSets
- Rolling updates
- Rollback capability
- Scaling
- Self-healing

Compose Equivalent:

```
deploy:
  replicas: 3
```

Deployment Manifest

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
      - name: nginx
        image: nginx:1.25-alpine
        resources:
          limits:
            cpu: 500m
            memory: 512Mi
```

Deployment Operations

Scaling:

```
$ kubectl scale deployment web --replicas=5
```

Updates:

```
$ kubectl set image deployment/web nginx=nginx:1.26-alpine
```

Rollback:

```
$ kubectl rollout history deployment/web
$ kubectl rollout undo deployment/web
```

Update Strategies

RollingUpdate (Default):

```
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1
    maxUnavailable: 1
```

Recreate:

```
strategy:
  type: Recreate
```

Part 1.5: Services

Service Types:

1. **ClusterIP** - Internal only (default)
2. **NodePort** - External access via node IP
3. **LoadBalancer** - Cloud load balancer
4. **Headless** - Direct pod access (StatefulSets)

Purpose: Stable networking for pods

Service Example

Docker Compose:

```yaml
services:
  web:
    ports:
      - "8080:80"
```

Kubernetes:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web
spec:
  selector:
    app: web
  ports:
  - port: 8080
    targetPort: 80
  type: ClusterIP
```

Service Discovery

DNS Resolution:

```
<service-name>.<namespace>.svc.cluster.local
```

Example:

```
$ kubectl run test --image=busybox -it --rm -- sh
/ $ wget -O- http://web:8080
/ $ wget -O- http://web.default.svc.cluster.local:8080
```

Docker Compose Equivalent:

Simply use service name as hostname

Part 1.6: ConfigMaps & Secrets

ConfigMaps: Non-sensitive configuration

Secrets: Sensitive data (base64 encoded)

Docker Compose:

```
environment:
  - DATABASE_URL=postgres://db:5432/db
  - SECRET_KEY=mysecret
```

ConfigMap Example

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  DATABASE_URL: "postgres://db:5432/db"
  LOG_LEVEL: "info"
```

Usage in Pod:

```yaml
containers:
- name: app
  envFrom:
  - configMapRef:
      name: app-config
```

Secret Example

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
stringData:
  SECRET_KEY: mysecret
  API_TOKEN: abc123xyz
```

Usage in Pod:

```yaml
containers:
- name: app
  env:
  - name: SECRET_KEY
    valueFrom:
      secretKeyRef:
        name: app-secret
        key: SECRET_KEY
```

Volume Mounts for Config

Mount ConfigMap as Files:

```yaml
volumeMounts:
- name: config
  mountPath: /etc/config
volumes:
- name: config
  configMap:
    name: app-config
```

Mount Secret as Files:

```yaml
volumeMounts:
- name: secret
  mountPath: /etc/secret
volumes:
- name: secret
  secret:
    secretName: app-secret
```

Part 1.7: Storage

Volume Types:

- **emptyDir** - Temporary, pod lifetime
- **hostPath** - Node filesystem (dev only)
- **PersistentVolume** - Cluster-level storage
- **PersistentVolumeClaim** - Request for storage
- **StorageClass** - Dynamic provisioning

Volume Comparison

Docker Compose:

```
volumes:
  - db-data:/var/lib/postgresql/data
volumes:
  db-data:
```

Kubernetes:

```
volumeMounts:
- name: db-data
  mountPath: /var/lib/postgresql/data
volumes:
- name: db-data
  persistentVolumeClaim:
    claimName: db-data-pvc
```

PersistentVolumeClaim

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: db-data-pvc
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: local-path
```

Access Modes:

- ReadWriteOnce (RWO) - Single node
- ReadWriteMany (RWX) - Multiple nodes
- ReadOnlyMany (ROX) - Multiple nodes, read-only

StatefulSets for Stateful Apps

Features:

- Stable pod identities
- Ordered deployment/scaling
- Stable network IDs
- Persistent storage per pod

Use Cases:

- Databases
- Message queues
- Distributed systems requiring stable identity

StatefulSet Example

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: database
spec:
  serviceName: database
  replicas: 1
  selector:
    matchLabels:
      app: database
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
  template:
    spec:
      containers:
      - name: postgres
        image: postgres:16-alpine
```

Part 1.8: Namespaces

Purpose:

- Logical cluster subdivision
- Resource isolation
- Access control boundaries
- Multi-tenancy

Default Namespaces:

- `default` - Default for user resources
- `kube-system` - Control plane components
- `kube-public` - Public resources
- `kube-node-lease` - Node heartbeats

Working with Namespaces

Create Namespace:

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: development
```

Use Namespace:

```
$ kubectl create -f manifest.yaml -n development
$ kubectl get pods -n development
```

Set Default:

```
$ kubectl config set-context --current --namespace=development
```

Resource Quotas

Limit namespace resources:

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: dev-quota
  namespace: development
spec:
  hard:
    requests.cpu: "4"
    requests.memory: 8Gi
    limits.cpu: "8"
    limits.memory: 16Gi
    pods: "10"
    services: "5"
```

Cross-Namespace Communication

DNS Format:

```
<service>.<namespace>.svc.cluster.local
```

Example:

```
# From frontend namespace to backend namespace
$ curl http://api.backend.svc.cluster.local:3000
```

NetworkPolicies can restrict cross-namespace access

Part 1.9: kubectl Essentials

Get Resources:

```
$ kubectl get pods
$ kubectl get pods -o wide
$ kubectl get pods -o yaml
$ kubectl get all
```

Describe Resources:

```
$ kubectl describe pod nginx
$ kubectl describe node kind-worker
```

kubectl - Logs & Exec

View Logs:

```
$ kubectl logs pod-name
$ kubectl logs pod-name -f  # Follow
$ kubectl logs pod-name -c container-name
$ kubectl logs -l app=nginx --all-containers
```

Execute Commands:

```
$ kubectl exec pod-name -- ls /etc
$ kubectl exec -it pod-name -- sh
```

kubectl - Apply & Delete

Apply Manifests:

```
$ kubectl apply -f manifest.yaml
$ kubectl apply -f ./directory/
$ kubectl apply -k ./kustomize/
```

Delete Resources:

```
$ kubectl delete pod nginx
$ kubectl delete -f manifest.yaml
$ kubectl delete all -l app=nginx
```

kubectl - JSONPath

Extract Specific Data:

```
# Pod names
$ kubectl get pods -o jsonpath='{.items[*].metadata.name}'

# Pod IPs
$ kubectl get pods -o jsonpath='{.items[*].status.podIP}'

# Custom columns
$ kubectl get pods -o custom-columns=NAME:.metadata.name,STATUS:.status.phase
```

k9s - Terminal UI

Launch:

```
$ k9s
```

Key Features:

- Real-time cluster view
- Resource navigation
- Log viewing
- Shell access
- Port forwarding
- YAML editing

k9s - Essential Shortcuts

Navigation:

- `:pods` - View pods
- `:deploy` - View deployments
- `:svc` - View services
- `:ns` - View/switch namespaces
- `/` - Filter resources

Actions:

- `d` - Describe
- `l` - Logs
- `s` - Shell
- `y` - View YAML
- `e` - Edit
- `ctrl-d` - Delete

Part 1.10: Manifest Best Practices

Organization:

- Separate files by resource type
- Use meaningful names
- Group related resources

Labeling:

```yaml
metadata:
  labels:
    app: web
    version: v1
    tier: frontend
    environment: production
```

Resource Management

Always Set:

```
resources:
  requests:
    cpu: 100m
    memory: 128Mi
  limits:
    cpu: 500m
    memory: 512Mi
```

Requests: Guaranteed resources

Limits: Maximum resources

Health Checks

Liveness Probe:

```
livenessProbe:
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 30
  periodSeconds: 10
```

Readiness Probe:

```
readinessProbe:
  httpGet:
    path: /ready
    port: 8080
  periodSeconds: 5
```

Security Best Practices

```
securityContext:
  runAsNonRoot: true
  runAsUser: 1000
  allowPrivilegeEscalation: false
  readOnlyRootFilesystem: true
  capabilities:
    drop:
    - ALL
```

Always:

- Run as non-root
- Use read-only filesystems where possible
- Drop unnecessary capabilities
- Use specific image tags (not `latest`)

Part 1.11: Final Integration Lab

Objective:

Convert 3-tier Docker Compose application to Kubernetes

Components:

- PostgreSQL database (StatefulSet + PVC)
- Backend API (Deployment + Service)
- Frontend web app (Deployment + Service)

Concepts Integrated:

- Namespaces, Secrets, ConfigMaps
- Services, Deployments, StatefulSets
- Persistent storage, Health checks

Docker Compose Source

```yaml
version: '3.8'
services:
  frontend:
    image: nginx:1.25-alpine
    ports: ["8080:80"]
    depends_on: [backend]

  backend:
    image: nginx:1.25-alpine
    environment:
      DATABASE_URL: postgres://postgres:secret@database:5432/mydb
    depends_on: [database]

  database:
    image: postgres:16-alpine
    environment:
      POSTGRES_PASSWORD: secret
    volumes:
      - db-data:/var/lib/postgresql/data
volumes:
  db-data:
```

Kubernetes Solution Overview

11 Manifest Files:

1. Namespace
2. Secret (database credentials)
3. PVC (database storage)
4. StatefulSet (PostgreSQL)
5. Service (database - headless)
6. ConfigMap (backend config)
7. Deployment (backend)
8. Service (backend - ClusterIP)
9. ConfigMap (frontend HTML)
10. Deployment (frontend)
11. Service (frontend - NodePort)

Key Conversion Points

Docker Compose:

- Single file
- Automatic networking
- Simple volumes
- Environment variables
- Service discovery via names

Kubernetes:

- Multiple manifests
- Explicit Services
- PVC + StorageClass
- ConfigMaps + Secrets
- DNS-based discovery

Lab Architecture

```
Internet
    |
    v
Frontend Service (NodePort :30080)
    |
    v
Frontend Deployment (3 replicas)
    |
    v
Backend Service (ClusterIP)
    |
    v
Backend Deployment (2 replicas)
    |
    v
Database Service (Headless)
    |
    v
Database StatefulSet (1 replica + PVC)
```

Deployment Order

```
# 1. Foundation
$ kubectl apply -f 01-namespace.yaml
$ kubectl apply -f 02-secret.yaml

# 2. Storage
$ kubectl apply -f 03-pvc.yaml

# 3. Database
$ kubectl apply -f 04-database-statefulset.yaml
$ kubectl apply -f 05-database-service.yaml

# 4. Backend
$ kubectl apply -f 06-backend-configmap.yaml
$ kubectl apply -f 07-backend-deployment.yaml
$ kubectl apply -f 08-backend-service.yaml

# 5. Frontend
$ kubectl apply -f 09-frontend-configmap.yaml
$ kubectl apply -f 10-frontend-deployment.yaml
$ kubectl apply -f 11-frontend-service.yaml
```

Verification

```
# Check all resources
$ kubectl get all -n final-lab

# Check pods are running
$ kubectl get pods -n final-lab

# Check services
$ kubectl get svc -n final-lab

# Test frontend
$ curl http://localhost:30080

# Check logs
$ kubectl logs -n final-lab -l app=backend
$ kubectl logs -n final-lab -l app=frontend
```

Common Patterns Learned

Configuration:

- ConfigMaps for non-sensitive data
- Secrets for credentials
- Environment variables vs. volume mounts

Networking:

- ClusterIP for internal services
- NodePort for development/testing
- DNS-based service discovery

Storage:

- PVCs for persistent data
- StatefulSets for stateful apps
- EmptyDir for temporary data

Best Practices Recap

1. **Always use labels** for organization
2. **Set resource requests/limits** on all containers
3. **Implement health checks** (liveness + readiness)
4. **Use namespaces** for logical separation
5. **Separate secrets** from configuration
6. **Use declarative manifests** (not imperative commands)
7. **Version your images** (avoid `latest`)
8. **Test in dev cluster** before production

Common Mistakes to Avoid

- Using `latest` image tag
- No resource limits (can starve cluster)
- Missing health checks (restart loops)
- Mixing config and secrets
- Not using labels consistently
- Hardcoding values (use ConfigMaps)
- Skipping readiness probes (traffic to non-ready pods)
- Forgetting to set namespaces

Troubleshooting Guide

Pod not starting:

```
$ kubectl describe pod <name>
$ kubectl logs <name>
$ kubectl get events --sort-by=.metadata.creationTimestamp
```

Service not accessible:

```
$ kubectl get svc
$ kubectl get endpoints <service>
$ kubectl describe svc <service>
```

Storage issues:

```
$ kubectl get pv,pvc
$ kubectl describe pvc <name>
```

Key Differences: Compose vs K8s

Aspect	Docker Compose	Kubernetes
Scope	Single host	Multi-node cluster
Config	One file	Multiple manifests
Scaling	Basic	Advanced + auto-scaling
Networking	Bridge networks	Services + DNS
Storage	Volumes	PV/PVC/StorageClass
Updates	Restart	Rolling updates
Health	Basic healthcheck	Liveness/Readiness/Startup
State	None	Controllers maintain desired state

Part 1 Summary

What We Covered:

1. Kubernetes architecture and concepts
2. Pods - smallest deployable units
3. Deployments - managing replicas and updates
4. Services - stable networking
5. ConfigMaps & Secrets - configuration management
6. Storage - volumes and persistence
7. Namespaces - logical organization
8. kubectl & k9s - essential tools
9. Manifest best practices
10. Complete application migration
11. Hands-on labs throughout

Resources Created Today

Learned to create:

- Pods
- Deployments (with rolling updates)
- Services (ClusterIP, NodePort, Headless)
- ConfigMaps (literals, files, volume mounts)
- Secrets (stringData, base64 data)
- PersistentVolumeClaims
- StatefulSets
- Namespaces
- ResourceQuotas

Commands Mastered

```
# Cluster management
kubectl cluster-info
kubectl get nodes

# Resource operations
kubectl apply -f manifest.yaml
kubectl get pods
kubectl describe pod <name>
kubectl logs <name>
kubectl exec -it <name> -- sh

# Debugging
kubectl get events
kubectl top nodes
kubectl top pods

# k9s TUI
k9s
```

Workshop Resources

Documentation:

- kubectl cheatsheet
- k9s shortcuts reference
- Docker Compose to K8s mapping guide

Examples:

- Complete examples for each concept
- Hands-on labs with solutions
- Final integration lab

Next Steps:

- Part 2: Advanced topics
- Practice on your own clusters
- Explore official Kubernetes docs

What's Next: Part 2

Advanced Topics (4 hours):

1. Ingress Controllers
2. Helm - Package Management
3. GitOps with Flux
4. Monitoring & Logging
5. Advanced Deployments (Blue/Green, Canary)
6. Autoscaling (HPA, VPA, Cluster Autoscaler)
7. Security & RBAC
8. Multi-cluster Management

Practice Recommendations

Between Part 1 and Part 2:

1. Deploy your own applications
2. Practice kubectl commands
3. Explore k9s features
4. Try different deployment strategies
5. Experiment with StatefulSets
6. Create your own labs
7. Review official documentation
8. Join Kubernetes community

Additional Learning Resources

Official:

- kubernetes.io/docs
- kubectl.docs.kubernetes.io
- k9scli.io

Community:

- CNCF Slack
- Kubernetes GitHub
- Stack Overflow

Practice:

- Play with Kubernetes (katacoda successor)
- KillerCoda scenarios
- Local kind/minikube clusters

Questions & Discussion

Common Questions:

- When to use Deployments vs StatefulSets?
- How to handle secrets securely?
- Best namespace organization strategy?
- Resource limits recommendations?
- Development vs Production differences?

Open Discussion:

- Share your use cases
- Challenges you anticipate
- What excites you about Kubernetes?

Feedback & Next Session

Today's Feedback:

- What worked well?
- What needs clarification?
- Pace okay?
- Lab difficulty appropriate?

Before Part 2:

- Complete any unfinished labs
- Review cheatsheets
- Practice kubectl commands
- Prepare questions

See you in Part 2!

Thank You!

Workshop Materials:

- All examples available in repository
- Labs with complete solutions
- Cheatsheets and reference guides

Support:

- GitHub issues for questions
- Workshop community discussions
- Office hours (if applicable)

Keep Learning!

Appendix: Quick Reference

Essential Commands:

```
# Apply manifests
kubectl apply -f file.yaml

# Get resources
kubectl get pods
kubectl get all

# Describe/logs
kubectl describe pod <name>
kubectl logs <name>

# Access cluster
kubectl port-forward pod/<name> 8080:80
kubectl exec -it <name> -- sh

# Cleanup
kubectl delete -f file.yaml
kubectl delete pod <name>
```

Appendix: Resource Template

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app
  labels:
    app: app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: app
  template:
    metadata:
      labels:
        app: app
    spec:
      containers:
      - name: app
        image: nginx:1.25-alpine
        ports:
        - containerPort: 80
        resources:
          requests:
            cpu: 100m
            memory: 128Mi
          limits:
            cpu: 500m
            memory: 512Mi
        livenessProbe:
          httpGet:
            path: /health
            port: 80
        readinessProbe:
          httpGet:
            path: /ready
            port: 80
```