Devops Shack

YAML GUIDE

BASIC TO ADVANCED

E-BOOK 2025

www.devopsshack.com

Click here for DevSecOps & Cloud DevOps Course

DevOps Shack

Basic to Advanced Kubernetes YAML Guide

Table of Contents

Part 3: Intermediate Kubernetes YAML Files

- Deployments
  - Scaling Applications with Replicas
  - Template and Selector Fields
  - Example YAML for a Deployment
- Services
  - Exposing Applications Within the Cluster
  - ClusterIP, NodePort, and LoadBalancer Types
  - Example YAML for a Service

Part 4: Adding Networking

- Ingress Resources
  - Introduction to Ingress
  - Routing Traffic to Services
  - Example YAML for Ingress Configuration
- DNS and Service Discovery in Kubernetes

Part 5: Advanced Kubernetes YAML

- Persistent Volumes (PV) and Persistent Volume Claims (PVC)
  - Storing Data Beyond Pod Lifecycles
  - Example YAML for PV and PVC
- ConfigMaps and Secrets
  - Managing Configuration Data
  - Storing Sensitive Data Securely
  - Example YAML for ConfigMaps and Secrets
- StatefulSets

    - Managing Stateful Applications
    - Example YAML for a StatefulSet
- Horizontal Pod Autoscaling
    - Autoscaling Based on CPU/Memory Usage
    - Example YAML for HPA
- Custom Resource Definitions (CRDs)
    - Extending Kubernetes with Custom Resources
    - Example YAML for a CRD

Part 6: Validating and Deploying Kubernetes YAML

- Validating YAML Syntax
    - Using kubectl for Validation
    - Common Validation Errors and Fixes
- Deploying YAML Files
    - Applying Configurations with kubectl
    - Verifying Resource Status
- Troubleshooting Deployment Issues

Part 7: Variations in Kubernetes YAML Files

- Multi-Container Pods
    - Sidecar Patterns
    - Example YAML for Multi-Container Pods
- Environment Variables in Pods
    - Injecting Configuration into Containers
    - Example YAML for Environment Variables
- Advanced Service Types

  - NodePort, LoadBalancer, and ClusterIP
  - Example YAML for Different Service Types
- Combining Resources in a Single File
  - Best Practices for Organization

Part 8: Example Project

- Overview of the Full-Stack Application
  - Architecture and Components
  - Features of the Application
- Resource Definitions
  - Frontend Deployment and Service
  - Backend Deployment and Service
  - Database StatefulSet and Service
  - Ingress Configuration
  - Horizontal Pod Autoscaler (HPA)
- Project Setup and Directory Structure
- Deployment Steps
  - Setting Up the Kubernetes Cluster
  - Applying YAML Files in Sequence
  - Verifying Resource Status
- Testing the Application
  - Verifying Database Connectivity
  - Testing Frontend and Backend Integration
  - Simulating Load for Autoscaling
- Accessing the Application
  - Using Ingress for External Traffic

Conclusion

- Summary of Learnings
- Importance of Kubernetes YAML in Real-World Applications
- Next Steps for Mastery

Introduction

In the dynamic world of software development and operations, Kubernetes has emerged as a critical tool for managing containerized applications at scale. Whether you are an individual developer deploying a small application or an enterprise managing thousands of microservices, Kubernetes provides the foundation to ensure reliability, scalability, and maintainability. This document is designed to take you on a comprehensive journey from the basics of Kubernetes YAML configurations to advanced use cases, culminating in the deployment of a fully-functional full-stack application.

Kubernetes employs declarative configurations written in YAML to define, manage, and automate the desired state of your resources, such as pods, deployments, services, and more. YAML (Yet Another Markup Language) is a human-readable format that ensures clarity and simplicity, making it easier to describe the configurations necessary for deploying and maintaining applications. This document delves into every critical aspect of writing Kubernetes YAML files, starting from the most basic structures and gradually advancing to more complex configurations that demonstrate the power of Kubernetes.

The first part of this guide introduces the fundamental building blocks of Kubernetes YAML files, offering step-by-step explanations of essential resource types like Pods, Deployments, and Services. These foundational elements help you understand how to containerize and deploy applications while enabling network communication within the Kubernetes cluster. Moving forward, you'll learn to handle more advanced scenarios such as persistent storage, environment variable injection, and managing stateful workloads.

One of the highlights of this document is the incorporation of advanced Kubernetes features, such as horizontal pod autoscaling, ingress for HTTP routing, and the use of ConfigMaps and Secrets to manage configuration and sensitive data. These elements not only showcase the flexibility of Kubernetes but also align with industry best practices to build resilient and secure applications.

As you progress, the document culminates in a hands-on project that ties together all the concepts discussed. The project involves deploying a full-stack web application, comprising a frontend, backend, and a database, within a Kubernetes cluster. This project demonstrates real-world scenarios where

various YAML configurations work together seamlessly to achieve a production-ready environment. The deployment showcases a robust setup that includes load balancing, horizontal scaling, persistent storage, and external ingress routing, offering a holistic view of how modern applications are deployed in Kubernetes.

This guide is tailored for beginners eager to learn Kubernetes YAML basics as well as seasoned professionals seeking to deepen their understanding of advanced Kubernetes configurations. With clear explanations, practical examples, and a focus on best practices, this document serves as a comprehensive resource for anyone looking to master Kubernetes YAML configurations and apply them effectively in real-world projects.

By the end of this document, you will have:

1. A deep understanding of Kubernetes YAML file structure and syntax.
2. The ability to define and deploy Kubernetes resources ranging from simple Pods to complex StatefulSets and Ingress configurations.
3. Practical knowledge of deploying a full-stack application with robust features like autoscaling, persistent storage, and secure networking.

Whether you are building applications for personal projects, startups, or large-scale enterprises, this guide will equip you with the knowledge and tools to leverage Kubernetes effectively. Let's embark on this journey to understand and master Kubernetes YAML files, from the basics to the advanced, ensuring that you are well-prepared to tackle the challenges of modern application deployment and orchestration.

Part 1: Basic YAML Concepts

Before diving into Kubernetes-specific YAML, understand the basics of YAML syntax:

1. **Key-Value Pairs**:
    - YAML uses key: value pairs for defining configurations.

```
name: my-app
version: 1.0
```

2. **Nested Structures**:
    - Indentation indicates hierarchy.

```
metadata:
  name: my-app
  labels:
    app: frontend
```

3. **Lists**:
    - Use dashes (-) for defining lists.

```
containers:
  - name: nginx
    image: nginx:latest
```

Part 2: Starting with Basic Kubernetes YAML Files

We begin with the simplest Kubernetes resources: Pods.

1. Writing a Pod YAML File

```
apiVersion: v1
kind: Pod
metadata:
  name: basic-pod
spec:
  containers:
    - name: nginx-container
```

```
    image: nginx:latest
```

Explanation:

- **apiVersion**: Defines the Kubernetes API version.
- **kind**: Specifies the resource type (Pod).
- **metadata**: Contains the resource name (basic-pod).
- **spec**: Describes the container within the Pod:
    - **name**: Name of the container.
    - **image**: Specifies the Docker image to use.

Part 3: Intermediate Kubernetes YAML Files

After understanding Pods, learn how to scale applications with Deployments.

2. Writing a Deployment YAML File

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: intermediate-deployment
spec:
 replicas: 2
 selector:
  matchLabels:
   app: nginx
 template:
  metadata:
   labels:
    app: nginx
  spec:
```

```
    containers:
      - name: nginx-container
        image: nginx:1.21.6
        ports:
          - containerPort: 80
```

Key Additions:

- **replicas**: Defines the number of Pods to run.
- **selector**: Matches labels for Pods managed by the Deployment.
- **template**:
    - **metadata**: Defines labels for Pods.
    - **spec**: Contains container details, including exposed ports.

Part 4: Adding Networking with Services

A Service allows external or internal access to your Pods.

3. Writing a Service YAML File

```
apiVersion: v1
kind: Service
metadata:
  name: intermediate-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

```
  type: LoadBalancer
```

Key Additions:

- **selector**: Matches Pods using the app: nginx label.
- **ports**:
  - port: The port exposed by the Service.
  - targetPort: The port the container listens to.
- **type**: Defines the Service type (ClusterIP, NodePort, or LoadBalancer).

Part 5: Advanced Kubernetes YAML

Now let's explore ConfigMaps, Secrets, and StatefulSets for more complex use cases.

4. ConfigMaps for Configuration

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  app.properties: |
    key1=value1
    key2=value2
```

Usage: Inject configuration data into Pods.

5. Secrets for Sensitive Data

```
apiVersion: v1
kind: Secret
metadata:
```

```
  name: db-secret
type: Opaque
data:
  username: YWRtaW4=  # base64 for "admin"
  password: cGFzc3dvcmQ=  # base64 for "password"
```

Usage: Securely store sensitive data.

6. StatefulSets for Stateful Applications

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: database
spec:
  serviceName: "db-service"
  replicas: 3
  selector:
    matchLabels:
      app: db
  template:
    metadata:
      labels:
        app: db
    spec:
      containers:
        - name: mysql
          image: mysql:5.7
```

```
      env:
       - name: MYSQL_ROOT_PASSWORD
         valueFrom:
           secretKeyRef:
             name: db-secret
             key: password
```

Advanced Features:

- StatefulSets provide unique identities to Pods.
- Uses Secrets for secure configuration.

Part 6: Validating and Deploying YAML Files

1. **Validate YAML Syntax**: Use tools like kubectl to validate:

```
kubectl apply --dry-run=client -f <file.yaml>
```

2. **Apply to Cluster**: Deploy the resource:

```
kubectl apply -f <file.yaml>
```

3. **Check Resource Status**:

```
kubectl get pods
kubectl get deployments
kubectl get services
```

Part 7: Variations in Kubernetes YAML Files

Below are various scenarios and their respective Kubernetes YAML variations:

1. Multiple Containers in a Pod

Pods can host multiple containers, often used for sidecar patterns (e.g., logging or monitoring containers alongside an application).

```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: multi-container-pod
spec:
  containers:
    - name: app-container
      image: nginx:latest
      ports:
        - containerPort: 80
    - name: sidecar-container
      image: busybox
      command: ["sh", "-c", "echo Sidecar logging container"]
```

Use Case: Logging, proxy, or monitoring sidecars.

2. Environment Variables in Pods

You can inject environment variables into containers.

```
apiVersion: v1
kind: Pod
metadata:
  name: env-var-pod
spec:
  containers:
    - name: app-container
      image: nginx:latest
      env:
        - name: ENV_NAME
```

```
        value: production
      - name: API_URL
        value: "http://api.example.com"
```

Use Case: Configure container behavior dynamically.

3. Using Persistent Volumes and Persistent Volume Claims

Persistent storage for stateful applications.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-example
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /data/pv-example
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-example
spec:
  accessModes:
    - ReadWriteOnce
```

```
  resources:
    requests:
      storage: 500Mi
---
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-pvc
spec:
  containers:
    - name: app-container
      image: nginx:latest
      volumeMounts:
        - name: storage
          mountPath: /data
  volumes:
    - name: storage
      persistentVolumeClaim:
        claimName: pvc-example
```

Use Case: Store data that persists beyond Pod lifecycles.

4. Advanced Service Types (e.g., Ingress)

Ingress enables external HTTP/S traffic to Services.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
```

```
  name: ingress-example
spec:
  rules:
    - host: example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: frontend-service
                port:
                  number: 80
```

Use Case: Route traffic based on domain names or URLs.

5. Horizontal Pod Autoscaler

Autoscale Deployments based on CPU or memory usage.

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-example
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: example-deployment
```

```
  minReplicas: 2
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

Use Case: Automatically adjust workloads during high traffic.

6. Custom Resource Definitions (CRDs)

Extend Kubernetes with custom resources.

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: databases.example.com
spec:
  group: example.com
  names:
    kind: Database
    listKind: DatabaseList
    plural: databases
    singular: database
  scope: Namespaced
  versions:
```

```
  - name: v1
    served: true
    storage: true
    schema:
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            properties:
              storageSize:
                type: string
```

Use Case: Build domain-specific resources.

Part 8: Example Project

Here's a project that incorporates all the concepts:

Objective

Deploy a full-stack application with:

1. Frontend
2. Backend

3. Database
4. Networking
5. Persistent Storage
6. Autoscaling

Project Setup Description

This project demonstrates how to deploy a **full-stack web application** on Kubernetes. The application consists of three primary components:

1. **Frontend**: A user interface built with a web framework (e.g., React or Angular).
2. **Backend**: A REST API or server-side application that serves data to the frontend.
3. **Database**: A persistent storage layer (e.g., MongoDB) for managing application data.

The project setup incorporates Kubernetes best practices, including deployments, services, persistent storage, ingress, and autoscaling.

Application Architecture

1. **Frontend**:
    - Serves static files (HTML, CSS, JavaScript).
    - Communicates with the backend for dynamic content.
2. **Backend**:
    - Handles business logic and API endpoints.
    - Communicates with the database for data retrieval and storage.
3. **Database**:
    - Provides persistent storage for the backend.

These components are connected using **Kubernetes Services**, while an **Ingress** exposes the frontend to external users.

Key Components and Their Roles

1. Frontend

- **Deployment**:
  - Runs a Docker container hosting the frontend application.
  - Configured to have 3 replicas for high availability.
- **Service**:
  - Exposes the frontend to the backend and ingress.
  - Uses a ClusterIP service type for internal communication.

2. Backend

- **Deployment**:
  - Runs a Docker container hosting the backend application.
  - Configured to have 2 replicas.
  - Injects the database connection URL via environment variables.
- **Service**:
  - Exposes the backend to the frontend and ingress.
  - Also uses a ClusterIP service type.

3. Database

- **StatefulSet**:
  - Ensures a stable identity for database Pods.
  - Uses a PersistentVolumeClaim (PVC) for storing data.
  - Configured with one replica for simplicity.
- **Service**:
  - Allows the backend to connect to the database using DNS-based discovery (database-service).

4. Networking

- **Ingress**:

- Routes external traffic to the appropriate service.
- Configured to expose:
  - Frontend (/ path).
  - Backend API (/api path).

5. Autoscaling

- **Horizontal Pod Autoscaler (HPA)**:
  - Configured for the backend deployment.
  - Automatically adjusts the number of Pods based on CPU usage (e.g., scales from 2 to 10 replicas).

Directory Structure

The project files are organized as follows:

```
├── frontend-deployment.yaml
├── backend-deployment.yaml
├── database-statefulset.yaml
├── frontend-service.yaml
├── backend-service.yaml
├── database-service.yaml
├── ingress.yaml
├── autoscaler.yaml
└── README.md
```

Features Included

1. **Persistence**:
   - The database uses a persistent volume, ensuring data durability.
2. **Scalability**:

- Backend deployment scales automatically based on CPU usage via HPA.

3. **Networking**:
   - Internal communication between components uses services.
   - External traffic is routed via ingress.
4. **Modular Deployment**:
   - Each component is defined separately, allowing for independent updates.

Testing

1. **Database Connectivity**:
   - Log into the backend Pod:

```
kubectl exec -it <backend-pod-name> -- sh
curl http://database-service:27017
```

2. **Frontend and Backend Integration**:
   - Test API calls from the frontend using browser dev tools or Postman.
3. **Scaling**:
   - Simulate high traffic and observe the backend HPA scaling the Pods.

1. Frontend Deployment and Service

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend-deployment
  labels:
```

```
    app: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
        - name: frontend-container
          image: frontend:latest
          ports:
            - containerPort: 3000
---
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  selector:
    app: frontend
  ports:
    - protocol: TCP
```

```
    port: 80
    targetPort: 3000
  type: ClusterIP
```

2. Backend Deployment and Service

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-deployment
  labels:
    app: backend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
        - name: backend-container
          image: backend:latest
          env:
            - name: DB_URL
```

```
          value: "mongodb://database-service:27017/mydb"
        ports:
          - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  selector:
    app: backend
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
  type: ClusterIP
```

3. Database StatefulSet and Service

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: database
spec:
  serviceName: "database-service"
  replicas: 1
  selector:
```

```
  matchLabels:
    app: database
 template:
  metadata:
    labels:
      app: database
  spec:
    containers:
      - name: mongo
        image: mongo:latest
        ports:
          - containerPort: 27017
        volumeMounts:
          - name: db-storage
            mountPath: /data/db
 volumeClaimTemplates:
  - metadata:
      name: db-storage
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 1Gi
---
apiVersion: v1
kind: Service
```

```
metadata:
  name: database-service
spec:
  selector:
    app: database
  ports:
    - protocol: TCP
      port: 27017
      targetPort: 27017
  type: ClusterIP
```

4. Ingress

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app-ingress
spec:
  rules:
    - host: myapp.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: frontend-service
```

```
            port:
              number: 80
      - path: /api
        pathType: Prefix
        backend:
          service:
            name: backend-service
            port:
              number: 8080
```

5. Horizontal Pod Autoscaler for Backend

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
 name: backend-hpa
spec:
 scaleTargetRef:
  apiVersion: apps/v1
  kind: Deployment
  name: backend-deployment
 minReplicas: 2
 maxReplicas: 10
 metrics:
  - type: Resource
    resource:
      name: cpu
```

```
    target:
      type: Utilization
      averageUtilization: 50
```

Deployment Process

Step 1: Prepare Kubernetes Cluster

Ensure you have a running Kubernetes cluster. If not:

- Use **Minikube** or **Kind** for local testing.
- Use **AWS EKS**, **Azure AKS**, or **GCP GKE** for cloud deployment.

Verify the cluster:

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

Step 2: Apply YAML Files

1. **Apply the database components**:

```
kubectl apply -f database-statefulset.yaml
kubectl apply -f database-service.yaml
```

2. **Apply the backend components**:

```
kubectl apply -f backend-deployment.yaml
kubectl apply -f backend-service.yaml
```

3. **Apply the frontend components**:

```
kubectl apply -f frontend-deployment.yaml
kubectl apply -f frontend-service.yaml
```

4. **Apply the ingress and autoscaler**:

```
kubectl apply -f ingress.yaml
kubectl apply -f autoscaler.yaml
```

Step 3: Verify Resources

Check the status of the deployed resources:

```
kubectl get all
```

- Pods: Verify all Pods are running.
- Services: Ensure services have been created with the correct ports.
- Ingress: Confirm the ingress is exposing the application.

Access the Application

1. **Ingress Host**:
    - Update your DNS to point myapp.example.com to your Kubernetes ingress controller IP.
    - Alternatively, use localhost or the ingress IP for local testing.
2. **Frontend**:
    - Access via the base path (/):

```
http://myapp.example.com/
```

3. **Backend API**:
    - Access via the /api path:

```
http://myapp.example.com/api
```

1. Access the application via the Ingress host (myapp.example.com).

This example covers a full-stack setup, integrating all concepts from basic to advanced Kubernetes YAML.

Conclusion

Kubernetes has revolutionized the way modern applications are deployed, managed, and scaled. YAML files serve as the backbone of Kubernetes configurations, offering a declarative way to define and maintain your application's infrastructure. By mastering Kubernetes YAML files, you unlock the ability to efficiently manage workloads, automate deployments, and implement robust application architectures.

This document has guided you from the basics of YAML syntax to the advanced features of Kubernetes, covering a wide range of topics such as Pods, Deployments, Services, Persistent Volumes, ConfigMaps, Secrets, Ingress, and Horizontal Pod Autoscaling. Each concept has been paired with practical examples and real-world use cases to ensure a hands-on learning experience.

The culmination of this guide was a full-stack application project that brought together everything you learned, providing a realistic scenario where multiple Kubernetes components work in harmony. From deploying individual resources like frontend and backend services to integrating features like scaling and persistent storage, this project demonstrates the versatility and power of Kubernetes.

Key Takeaways

1. **Foundation in YAML and Kubernetes**: You now have a solid understanding of YAML syntax and how it translates into Kubernetes resource configurations.
2. **Building Scalable Applications**: You've learned to deploy and scale applications using Deployments, Services, and Horizontal Pod Autoscaling.
3. **Advanced Configurations**: You've explored features like StatefulSets, Ingress, and Persistent Volumes, equipping you with the knowledge to manage complex workloads.
4. **End-to-End Application Management**: The hands-on project demonstrated how to design, deploy, and manage a multi-tier application in Kubernetes.

Next Steps

- **Experiment with Custom Scenarios**: Try creating your own projects with different configurations, such as multi-tenant applications or CI/CD pipelines.
- **Explore Kubernetes Ecosystem Tools**: Learn about Helm for managing application packages, Prometheus for monitoring, and Istio for service mesh.
- **Dive Deeper into Cloud-Native Concepts**: Understand how Kubernetes integrates with cloud providers like AWS, Azure, and Google Cloud.
- **Contribute to the Community**: Engage with the Kubernetes community by contributing to open-source projects or sharing your knowledge through blogs or videos.

Mastering Kubernetes YAML files is just the beginning of your journey into the cloud-native world. With these skills, you are well-prepared to tackle real-world challenges, design resilient systems, and scale applications effectively. As you continue to explore Kubernetes, you will uncover its immense potential to transform how software is built and delivered.

Kubernetes empowers you to build for the future, and this guide is your first step toward becoming a cloud-native expert. Keep experimenting, keep learning, and embrace the limitless possibilities of Kubernetes!