## Docker Structure and Its Components

Docker is an open-source platform designed for developing, shipping, and running applications in lightweight containers. It employs a client-server architecture, which allows developers to package applications with all their dependencies into standardized units. Below is a detailed overview of Docker's structure and its key components.

## Key Components of Docker

1. Docker Engine
   * Definition: The core component of Docker, responsible for building, running, and managing containers.
   * Subcomponents:
     + Docker Daemon (dockerd): A persistent background process that listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. It processes commands from the Docker client and can communicate with other daemons to manage services

.

* + - REST API: This API allows applications to interact with the Docker daemon. It can be accessed via HTTP clients and is crucial for executing commands programmatically

.

* + - Command Line Interface (CLI): The primary tool for users to interact with Docker. Commands like docker run are executed through the CLI, which sends instructions to the daemon

1. Docker Images
   * Definition: Read-only templates used to create containers. They include everything needed to run an application—code, libraries, environment variables, and configuration files.
   * Creation: Images are built from a set of instructions defined in a Dockerfile. Each instruction in the Dockerfile creates a new layer in the image, making them efficient and quick to deploy.
2. Dockerfile
   * Definition: A script containing a series of instructions on how to build a Docker image.
   * Contents: It specifies the base image, environment variables, commands to run during the build process, and any additional files needed for the application. The commands are executed in sequence to create a layered image structure

.

1. Docker Containers
   * Definition: Runnable instances of Docker images. Unlike images, containers are mutable and can be modified during runtime.
   * Functionality: Containers provide an isolated environment for applications to run without interference from other processes or containers on the host machine. When a container is removed, any changes made that are not saved persistently are lost
2. Docker Hub
   * Definition: A cloud-based registry service for sharing Docker images.
   * Functionality: Users can upload their images for public access or download images created by others. It simplifies collaboration among developers by providing a centralized repository for container images
3. Docker Volumes
   * Definition: Persistent storage mechanisms that allow data to persist beyond the lifecycle of a container.
   * Usage: Volumes are stored outside the container's file system and can be shared between multiple containers, making them ideal for storing application data that needs to be retained even after container termination.
4. Docker Compose
   * Definition: A tool for defining and running multi-container Docker applications.
   * Functionality: Using a docker-compose.yml file, developers can specify how multiple containers should interact within an application stack, simplifying deployment and management
5. Networking in Docker
   * Definition: Mechanisms that allow containers to communicate with each other and with external systems.
   * Types of Networks:
     + Bridge Network: The default network type that allows containers on the same host to communicate.
     + Overlay Network: Enables communication between containers across different hosts in a distributed system.
     + Macvlan Network: Allows assigning MAC addresses to containers so they appear as physical devices on the network.

## Detailed Process Overview

* When using Docker, developers typically begin by writing a Dockerfile that defines how their application should be built into an image.
* The docker build command is then executed to create an image from this Dockerfile, generating layers based on each instruction provided.
* Once the image is ready, it can be run as a container using the docker run command. This command creates an instance of the image where the application runs in isolation from other processes.
* If persistent data storage is needed, volumes can be defined in either the Dockerfile or during container creation using flags.
* For applications requiring multiple services (e.g., web server, database), developers can define these services in a docker-compose.yml file and use docker-compose up to start them simultaneously.

By leveraging these components effectively, Docker enables streamlined development workflows, efficient resource utilization, and simplified deployment processes across various environments.

To create a Dockerfile for a Java Spring Boot application that uses Thymeleaf and PostgreSQL, you need to define the necessary instructions to build and run your application in a Docker container. Below is an example Dockerfile along with explanations for each part.

## Example Dockerfile

text

# Use the official OpenJDK image as the base image  
FROM openjdk:11-jre-slim  
  
# Set the working directory inside the container  
WORKDIR /app  
  
# Copy the JAR file from the target directory to the container  
COPY target/my-spring-boot-app.jar app.jar  
  
# Expose the port on which the application will run  
EXPOSE 8080  
  
# Set environment variables for PostgreSQL connection  
ENV SPRING\_DATASOURCE\_URL=jdbc:postgresql://db:5432/mydatabase  
ENV SPRING\_DATASOURCE\_USERNAME=myuser  
ENV SPRING\_DATASOURCE\_PASSWORD=mypassword  
  
# Command to run the application  
ENTRYPOINT ["java", "-jar", "app.jar"]

## Explanation of Each Instruction

1. FROM openjdk:11-jre-slim:
   * This line specifies the base image for your application. Here, we are using a lightweight version of OpenJDK 11, which is suitable for running Java applications.
2. WORKDIR /app:
   * This sets the working directory inside the container to /app. All subsequent commands will be executed in this directory.
3. COPY target/my-spring-boot-app.jar app.jar:
   * This command copies the built JAR file of your Spring Boot application from your local target directory into the /app directory in the container. Replace my-spring-boot-app.jar with the actual name of your JAR file.
4. EXPOSE 8080:
   * This informs Docker that the container listens on port 8080 at runtime. This is important for mapping ports when running the container.
5. ENV SPRING\_DATASOURCE\_URL=jdbc:postgresql://db:5432/mydatabase:
   * This sets an environment variable for connecting to a PostgreSQL database. The db refers to a service name if you're using Docker Compose (assuming PostgreSQL runs in another container).
6. ENV SPRING\_DATASOURCE\_USERNAME=myuser:
   * This sets the username for connecting to your PostgreSQL database.
7. ENV SPRING\_DATASOURCE\_PASSWORD=mypassword:
   * This sets the password for connecting to your PostgreSQL database.
8. ENTRYPOINT ["java", "-jar", "app.jar"]:
   * This specifies the command that will be executed when the container starts. It runs your Spring Boot application using Java.

## Building and Running Your Docker Container

To build and run your Docker container, follow these steps:

1. Build your Spring Boot application (ensure you have Maven installed):

Bash

mvn clean package

1. Build the Docker image (run this command in the directory containing your Dockerfile):

Bash

docker build -t my-spring-boot-app .

1. Run your Docker container, ensuring you link it with a PostgreSQL container if needed:

bash

docker run -d -p 8080:8080 --name my-app --link postgres-container:db my-spring-boot-app

In this command, replace postgres-container with the name of your PostgreSQL container.This setup allows you to run a Spring Boot application with Thymeleaf and connect it to a PostgreSQL database within a Docker environment efficiently.

To create a docker-compose.yml file for a Java Spring Boot application using Thymeleaf and PostgreSQL, you can define the services for both the Spring Boot application and the PostgreSQL database. Below is an example configuration that illustrates how to set this up.

Example docker-compose.yml

text

version: '3.8'  
  
services:  
 spring-app:  
 build:  
 context: .  
 dockerfile: Dockerfile  
 container\_name: spring-app  
 ports:  
 - "8080:8080"  
 environment:  
 SPRING\_DATASOURCE\_URL: jdbc:postgresql://db:5432/mydatabase  
 SPRING\_DATASOURCE\_USERNAME: myuser  
 SPRING\_DATASOURCE\_PASSWORD: mypassword  
 depends\_on:  
 - db  
  
 db:  
 image: postgres:13  
 container\_name: postgres-db  
 environment:  
 POSTGRES\_DB: mydatabase  
 POSTGRES\_USER: myuser  
 POSTGRES\_PASSWORD: mypassword  
 ports:  
 - "5432:5432"  
 volumes:  
 - postgres\_data:/var/lib/postgresql/data  
  
volumes:  
 postgres\_data:

Explanation of Each Section

version:

Specifies the version of the Docker Compose file format being used. Here, 3.8 is chosen for compatibility with most features.

services:

This section defines the different services (containers) that will be part of your application.

spring-app:

build: Specifies how to build the Spring Boot application image. It uses the current directory (.) and looks for a Dockerfile.

container\_name: Sets a custom name for the Spring Boot container.

ports: Maps port 8080 on the host to port 8080 in the container, allowing access to the application.

environment: Defines environment variables needed for the Spring Boot application to connect to PostgreSQL.

SPRING\_DATASOURCE\_URL: Connection string for PostgreSQL, using the service name db as the host.

SPRING\_DATASOURCE\_USERNAME and SPRING\_DATASOURCE\_PASSWORD: Credentials for accessing the database.

depends\_on: Ensures that the Spring Boot application starts only after the PostgreSQL database is up and running.

db:

image: Uses the official PostgreSQL image from Docker Hub.

container\_name: Sets a custom name for the PostgreSQL container.

environment: Configures PostgreSQL with a database name, user, and password.

POSTGRES\_DB: The name of the database to create.

POSTGRES\_USER and POSTGRES\_PASSWORD: Credentials for accessing PostgreSQL.

ports: Maps port 5432 on the host to port 5432 in the container, allowing access to PostgreSQL from outside.

volumes: Persists data in PostgreSQL across container restarts by using a named volume (postgres\_data).

volumes:

Defines a named volume (postgres\_data) that will store PostgreSQL data persistently.

Building and Running Your Application

To run your application using this Docker Compose configuration, follow these steps:

Ensure you have a valid Dockerfile in your project directory that builds your Spring Boot application.

Run Docker Compose to start both services:

bash

docker-compose up --build

Access your Spring Boot application at http://localhost:8080.

To stop and remove all containers, networks, and volumes defined in your Compose file, use:

bash

docker-compose down

This setup provides a straightforward way to run a Spring Boot application with Thymeleaf and PostgreSQL in Docker containers, facilitating development and deployment processes effectively.

Kubernetes is a powerful open-source platform designed for automating the deployment, scaling, and management of containerized applications. Its architecture is modular and consists of various components that work together to manage clusters effectively. Below is a detailed explanation of Kubernetes architecture and its key components.

## Kubernetes Architecture Overview

Kubernetes architecture can be divided into two main planes:

1. Control Plane: This manages the overall state of the cluster.
2. Data Plane: This consists of the worker nodes that run the containerized applications.

## Key Components of Kubernetes

## Control Plane Components

The control plane is responsible for managing the cluster and maintaining its desired state. It consists of several key components:

* kube-apiserver:
  + Acts as the central management point for the Kubernetes cluster.
  + Exposes the Kubernetes API, allowing external clients to interact with the cluster.
  + Validates and processes API requests, ensuring that only authorized users can access the cluster's resources.
* etcd:
  + A distributed key-value store used for storing all cluster data, including configuration data and state information.
  + Provides a reliable way to store data across distributed systems, ensuring consistency and availability.
* kube-scheduler:
  + Responsible for scheduling pods onto available nodes based on resource requirements and constraints.
  + Evaluates which nodes are suitable for new pods based on factors like resource availability, affinity rules, and taints/tolerations.
* kube-controller-manager:
  + Runs controllers that manage the state of the cluster by monitoring its current state and making adjustments as needed.
  + Includes various controllers such as replication controllers (to ensure desired replicas of pods) and node controllers (to monitor node health)
* cloud-controller-manager (optional):
  + Integrates with cloud service providers to manage cloud-specific resources such as load balancers or storage volumes

[2](https://kubernetes.io/docs/concepts/overview/components/)

.

## Node Components

Each node in a Kubernetes cluster runs specific components that facilitate the execution of workloads:

* kubelet:
  + An agent that runs on each node, responsible for managing pods and their containers.
  + Communicates with the kube-apiserver to receive instructions about which pods to run and reports back on their status.
* kube-proxy:
  + Maintains network rules on nodes to enable communication between services and pods.
  + Implements load balancing and routing for network traffic directed at services.
* Container Runtime:
  + The software is responsible for running containers within pods. Popular container runtimes include Docker, containerd, and CRI-O.

## Additional Components

Kubernetes also includes various add-ons that enhance its functionality:

* DNS: Provides service discovery within the cluster by resolving service names to IP addresses.
* Dashboard: A web-based UI for managing Kubernetes clusters.
* Monitoring Tools: Collect metrics and logs from containers to help in performance analysis and troubleshooting.

## Summary

Kubernetes architecture is designed for flexibility and scalability, allowing it to efficiently manage containerized applications across clusters. The separation into control plane and data plane components enables effective management while ensuring high availability and resilience. Each component plays a crucial role in maintaining the desired state of applications, scaling workloads, and providing networking capabilities within a Kubernetes environment.

Kubernetes offers various deployment strategies to manage application updates effectively while minimizing downtime and ensuring service reliability. Here are the key deployment strategies commonly used in Kubernetes:

## 1. Rolling Update

* Description: This is the default deployment strategy in Kubernetes. It gradually replaces old pods with new ones, ensuring that some instances of the application remain available during the update.
* Advantages:
  + Minimal downtime as new pods are incrementally rolled out.
  + Easy to roll back if issues arise since the previous version remains active until the update is confirmed.
* Implementation: Specify parameters like maxSurge (the number of additional pods that can be created during the update) and maxUnavailable (the number of pods that can be unavailable during the update) in the deployment configuration.

## 2. Blue/Green Deployment

* Description: This strategy involves maintaining two identical environments: "Blue" for the current production version and "Green" for the new version. Traffic is switched from Blue to Green once the new version is ready and tested.
* Advantages:
  + Instant rollback capability if issues occur after switching traffic.
  + Minimal downtime since both environments are live, allowing for thorough testing of the new version before going live.
* Implementation: Deploy both versions and use a service to route traffic between them. Switch traffic to Green after successful testing.

## 3. Canary Deployment

* Description: In a canary deployment, a new version of an application is rolled out to a small subset of users before a full-scale rollout. This allows for real-world testing with minimal risk.
* Advantages:
  + Reduces risk by exposing only a small percentage of users to the new version initially.
  + Allows for monitoring and gathering feedback before wider deployment.
* Implementation: Deploy two nearly identical versions (old and new) and gradually increase traffic to the new version based on performance metrics.

## 4. Recreate Deployment

* Description: This strategy involves shutting down all instances of the old version before deploying the new version. It results in downtime during the transition.
* Advantages:
  + Simplicity in implementation, as it does not require managing multiple versions simultaneously.
* Disadvantages:
  + Downtime is unavoidable, which may not be acceptable for many applications.

## 5. Ramped Slow Rollout

* Description: Similar to rolling updates, this strategy allows for controlled updates by specifying how many replicas can be updated at a time. It gradually increases the number of updated pods while ensuring that no pods become unavailable.
* Advantages:
  + Provides fine-grained control over rollout speed, reducing risks associated with updates.
* Implementation: Set parameters like maxSurge and maxUnavailable to control how many pods are updated at once.

## 6. Shadow Deployment

* Description: In shadow deployments, a new version of an application runs alongside the current version but does not receive user traffic directly. Instead, incoming requests are duplicated, sending one copy to each version for testing under real-world conditions.
* Advantages:
  + Allows teams to assess performance without impacting end-users.
  + Useful for testing non-functional aspects like load handling and stability.
* Disadvantages:
  + Requires additional resources since both versions run simultaneously.

## Summary

Choosing the right Kubernetes deployment strategy depends on various factors such as application architecture, user tolerance for downtime, and organizational capacity for handling rollbacks. Each strategy offers unique benefits and challenges, making it essential to tailor your approach based on specific requirements and scenarios.

To set up your Kubernetes environment using kubectl, follow these step-by-step commands. This guide assumes you have a Kubernetes cluster already set up and that you need to configure kubectl to interact with it.

## Step 1: Install kubectl

1. Download the latest kubectl binary:  
   Depending on your operating system, use the following command to download kubectl. For example, for macOS:

bash

curl -LO "<https://dl.k8s.io/release/$(curl> -L -s <https://dl.k8s.io/release/stable.txt)/bin/darwin/amd64/kubectl>"

1. Make the binary executable:

bash

chmod +x ./kubectl

1. Move the binary to your PATH:

bash

sudo mv ./kubectl /usr/local/bin/kubectl

1. Verify the installation:

bash

kubectl version --client

## Step 2: Set Up Configuration

1. Create the .kube directory:

bash

mkdir -p $HOME/.kube

1. Copy the Kubernetes configuration file:  
   Assuming you have access to the admin.conf file on your master node, run:

bash

sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

1. Change ownership of the config file:

bash

sudo chown $(id -u):$(id -g) $HOME/.kube/config

1. Set the KUBECONFIG environment variable (optional but recommended):

bash

export KUBECONFIG=$HOME/.kube/config

To make this change permanent, add it to your .bashrc or .bash\_profile:

bash

echo 'export KUBECONFIG=$HOME/.kube/config' >> $HOME/.bashrc  
source $HOME/.bashrc

## Step 3: Verify Cluster Access

1. Check cluster information:

bash

kubectl cluster-info

1. List all nodes in the cluster:

bash

kubectl get nodes

1. Check the status of pods in the kube-system namespace (to verify that essential services are running):

bash

kubectl get pods --namespace=kube-system

## Step 4: Configure Contexts (if needed)

If you are working with multiple clusters or contexts, you can manage them using kubectl config commands:

1. View current context:

bash

kubectl config current-context

1. List all contexts:

bash

kubectl config get-contexts

1. Switch to a different context:

bash

kubectl config use-context <context-name>

## Step 5: Deploy an Application (Example)

1. Create a deployment YAML file (e.g., nginx-deployment.yaml):

text

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: nginx-deployment  
 labels:  
 app: nginx  
spec:  
 replicas: 3  
 selector:  
 matchLabels:  
 app: nginx  
 template:  
 metadata:  
 labels:  
 app: nginx  
 spec:  
 containers:  
 - name: nginx  
 image: nginx:1.14.2  
 ports:  
 - containerPort: 80

1. Apply the deployment configuration:

bash

kubectl apply -f nginx-deployment.yaml

1. Check the status of the deployment:

bash

kubectl get deployments

1. Verify that the pods are running:

bash

kubectl get pods --show-labels

## Summary

Following these steps will set up your Kubernetes environment using kubectl, allowing you to manage your cluster and deploy applications effectively. Make sure to adjust commands based on your specific configuration and requirements, especially when dealing with multiple clusters or namespaces.

To set up Kubernetes Ingress with Envoy as the load balancer, follow these step-by-step instructions. This guide assumes you have a Kubernetes cluster running and kubectl configured.

## Step 1: Install Envoy Ingress Controller

1. Create a Namespace for Envoy:

bash

kubectl create namespace envoy

1. Deploy the Envoy Ingress Controller:  
   You can deploy the Envoy ingress controller using a YAML file. Create a file named envoy-ingress-controller.yaml with the following content:

text

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: envoy-ingress  
 namespace: envoy  
spec:  
 replicas: 1  
 selector:  
 matchLabels:  
 app: envoy-ingress  
 template:  
 metadata:  
 labels:  
 app: envoy-ingress  
 spec:  
 containers:  
 - name: envoy-ingress  
 image: envoyproxy/envoy:v1.21.0  
 ports:  
 - containerPort: 80  
 - containerPort: 443  
 volumeMounts:  
 - name: config-volume  
 mountPath: /etc/envoy  
 volumes:  
 - name: config-volume  
 configMap:  
 name: envoy-config  
  
---  
  
apiVersion: v1  
kind: Service  
metadata:  
 name: envoy-ingress-service  
 namespace: envoy  
spec:  
 type: LoadBalancer  
 ports:  
 - port: 80  
 targetPort: 80  
 - port: 443  
 targetPort: 443  
 selector:  
 app: envoy-ingress

1. Create a ConfigMap for Envoy Configuration:  
   Create a file named envoy-config.yaml with your desired configuration. Here’s a basic example:

text

apiVersion: v1  
kind: ConfigMap  
metadata:  
 name: envoy-config  
 namespace: envoy  
data:  
 envoy.yaml: |  
 static\_resources:  
 listeners:  
 - name: listener\_0  
 address:  
 socket\_address: { address: 0.0.0.0, port\_value: 80 }  
 filter\_chains:  
 - filters:  
 - name: "envoy.filters.network.http\_connection\_manager"  
 config:  
 codec\_type: AUTO  
 stat\_prefix: ingress\_http  
 route\_config:  
 name: local\_route  
 virtual\_hosts:  
 - name: backend\_service  
 domains: ["\*"]  
 routes:  
 - match: { prefix : "/" }  
 route:  
 cluster: backend\_service\_cluster  
  
 clusters:  
 - name: backend\_service\_cluster  
 connect\_timeout: 0.25s  
 type: STRICT\_DNS  
 lb\_policy: ROUND\_ROBIN  
 load\_assignment:  
 cluster\_name: backend\_service\_cluster  
 endpoints:  
 - lb\_endpoints:  
 - endpoint:  
 address:  
 socket\_address: { address: backend-service, port\_value: 80 }

1. Apply the Configurations:  
   Deploy the ConfigMap and the Ingress controller to your cluster:

bash

kubectl apply -f envoy-config.yaml  
kubectl apply -f envoy-ingress-controller.yaml

## Step 2: Deploy Your Application

1. Create a Sample Application Deployment (e.g., a simple NGINX application):Create a file named nginx-deployment.yaml:

text

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: backend-service  
 labels:  
 app: backend-service  
spec:  
 replicas: 2  
 selector:  
 matchLabels:  
 app: backend-service  
 template:  
 metadata:  
 labels:  
 app: backend-service  
 spec:  
 containers:  
 - name: nginx-backend  
 image: nginx   
 ports:  
 - containerPort: 80

1. Create a Service for Your Application:Create a file named nginx-service.yaml:

text

apiVersion: v1   
kind: Service   
metadata:  
 name: backend-service   
 labels:  
 app: backend-service   
spec:  
 ports:  
 - port : 80   
 targetPort : 80   
 selector :  
 app : backend-service

1. Apply the Application Deployment and Service:

bash

kubectl apply -f nginx-deployment.yaml   
kubectl apply -f nginx-service.yaml

## Step 3: Create an Ingress Resource

1. Define an Ingress Resource:

Create a file named envoy-ingress.yaml:

text

apiVersion : networking.k8s.io/v1   
kind : Ingress   
metadata :   
 name : envoy-ingress-resource   
 namespace : envoy   
spec :   
 rules :  
 - host : your-app.example.com # Change this to your domain or IP   
 http :  
 paths :  
 - path : /   
 pathType : Prefix   
 backend :  
 service :  
 name : backend-service   
 port :  
 number : 80

1. Apply the Ingress Resource:

bash

kubectl apply -f envoy-ingress.yaml

## Step 4: Verify Setup

1. Check the Status of Your Ingress Controller:

bash

kubectl get pods -n envoy

1. Get External IP of the Envoy Service:

bash

kubectl get service envoy-ingress-service --namespace=envoy

1. Test Access to Your Application:

Use curl or your browser to access the application through the configured host (e.g., <http://your-app.example.com>).

## Summary

By following these steps, you have successfully set up an Envoy load balancer as an Ingress controller in your Kubernetes environment, routing traffic to your applications based on defined rules. Adjust configurations as necessary to fit your specific use case and requirements

## Step-by-Step Instructions for Using Containerd, Helm Chart, and Kubernetes for Deployment

This guide provides a comprehensive approach to deploying applications using Containerd as the container runtime, Helm for managing Kubernetes applications, and Kubernetes as the orchestration platform.

## Prerequisites

* Kubernetes Cluster: Set up a Kubernetes cluster using tools like Minikube, Kind, or a cloud provider (AWS, GCP, Azure).
* Containerd: Ensure Containerd is installed and configured as the container runtime.
* Helm: Install Helm by following the installation guide on the [Helm website](https://helm.sh/docs/intro/install/).
* kubectl: Install kubectl to interact with your Kubernetes cluster.

## Step 1: Configure Containerd

1. Install Containerd:  
   Follow the installation instructions specific to your operating system. For example, on Ubuntu:

bash

sudo apt-get install containerd

1. Configure Containerd:  
   Create or edit the configuration file at /etc/containerd/config.toml to set parameters like the snapshotter and runtime options.

text

[plugins]  
 [plugins."io.containerd.grpc.v1.cri"]  
 ...

1. Restart Containerd:  
   After making changes, restart the service:

bash

sudo systemctl restart containerd

## Step 2: Create a Docker Image

1. Create a Dockerfile:  
   Create a simple Dockerfile for your application. For example, if using Nginx:

text

FROM nginx:alpine  
COPY index.html /usr/share/nginx/html

1. Build the Docker Image:  
   Build your image using Docker:

bash

docker build -t <username>/my-nginx:latest .

1. Push to a Container Registry:  
   Log in and push your image to Docker Hub or another registry:

bash

docker login  
docker push <username>/my-nginx:latest

## Step 3: Create a Helm Chart

1. Create a New Helm Chart:  
   Generate a new Helm chart for your application:

bash

helm create my-nginx-chart

1. Edit values.yaml:  
   Update values.yaml to specify your image details:

text

image:  
 repository: <username>/my-nginx  
 tag: latest  
 pullPolicy: IfNotPresent

1. Define Kubernetes Resources:  
   Modify the templates in templates/ directory (e.g., deployment.yaml, service.yaml) to include your application specifications.

## Step 4: Deploy Using Helm

1. Install Your Helm Chart:  
   Deploy your application using Helm with the following command:

bash

helm install my-release ./my-nginx-chart

1. Check Deployment Status:  
   Verify that your application is running by checking pod statuses:

bash

kubectl get pods

## Step 5: Expose Your Application

1. Create a Service:  
   Ensure you have a service defined in your Helm chart (service.yaml) to expose your application:

text

apiVersion: v1  
 kind: Service  
 metadata:  
 name: my-nginx-service  
 spec:  
 type: LoadBalancer # or NodePort depending on access needs  
 ports:  
 - port: 80  
 targetPort: 80  
 selector:  
 app: my-nginx-chart # Adjust according to your labels

1. Access Your Application:  
   Once the service is created, use kubectl get services to find the external IP or port where you can access your application.

## Conclusion

By following these steps, you can successfully deploy an application using Containerd as the container runtime, manage it with Helm charts, and orchestrate everything through Kubernetes. This process streamlines deployment and management of containerized applications in a scalable environment.

To install the ELK stack (Elasticsearch, Logstash, and Kibana) and configure them to work together, follow these structured steps:

## Prerequisites

* Ensure you have Java installed (JDK 8 or higher is recommended).
* Ensure your system has sufficient resources (at least 4GB of RAM).

## Step 1: Install Elasticsearch

1. Download Elasticsearch:
   * Visit the [official Elasticsearch download page](https://www.elastic.co/downloads/elasticsearch) and download the appropriate version for your operating system.
2. Install Elasticsearch:
   * Extract the downloaded archive and move it to your desired location.
   * For RPM-based systems, you can use:

bash

rpm --import <https://artifacts.elastic.co/GPG-KEY-elasticsearch>

Then create a repository file and install:

bash

sudo yum install elasticsearch

1. Configure Elasticsearch:
   * Edit the configuration file located at config/elasticsearch.yml to set parameters like cluster.name and network.host.
   * Example configuration:

text

network.host: localhost

1. Start Elasticsearch:
   * Use the following command to start Elasticsearch:

bash

sudo service elasticsearch start

## Step 2: Install Logstash

1. Download Logstash:
   * Visit the [official Logstash download page](https://www.elastic.co/downloads/logstash) and download the appropriate version.
2. Install Logstash:
   * Extract the downloaded files to a local directory.
3. Configure Logstash:
   * Create a configuration file named logstash-simple.conf in the config directory:

ruby

input { stdin { } }  
output {  
 elasticsearch {  
 hosts => ["<http://localhost:9200>"]  
 index => "my\_index"  
 }  
 stdout { codec => rubydebug }  
}

1. Start Logstash:
   * Navigate to the Logstash bin directory and execute:

bash

./logstash -f path/to/logstash-simple.conf

## Step 3: Install Kibana

1. Download Kibana:
   * Visit the [official Kibana download page](https://www.elastic.co/downloads/kibana) and download the appropriate version.
2. Install Kibana:
   * Extract the downloaded files to a local directory.
3. Configure Kibana:
   * Edit the configuration file located at config/kibana.yml to specify the Elasticsearch URL:

text

server.port: 5601  
elasticsearch.hosts: ["<http://localhost:9200>"]

1. Start Kibana:
   * Navigate to the Kibana bin directory and execute:

bash

./kibana

* + Verify that Kibana is running by visiting [http://localhost:5601](http://localhost:5601/) in your web browser.

## Step 4: Verify Installation

* After starting all components, ensure they are communicating correctly by checking Kibana's interface for available indices.
* You should see logs being ingested into Elasticsearch via Logstash.

## Additional Configuration

* For production environments, consider securing your ELK stack with SSL/TLS and setting up user authentication.
* Monitor resource usage and optimize settings based on your specific data volume and query patterns.

By following these steps, you will have a fully functional ELK stack set up on your machine, ready for log analysis and visualization.

To set up and configure the EFK stack (Elasticsearch, Fluentd, and Kibana) on Kubernetes, follow these step-by-step instructions. This guide will help you deploy each component and ensure they work together for effective log management.

## Step 1: Prepare Your Environment

1. Install Kubernetes: Ensure you have a running Kubernetes cluster. You can use Minikube, kind, or any cloud provider.
2. Install kubectl: Make sure you have kubectl installed and configured to interact with your cluster.

## Step 2: Create a Namespace

Creating a dedicated namespace helps organize resources:

bash

kubectl create namespace kube-logging

## Step 3: Deploy Elasticsearch

Elasticsearch will be deployed as a StatefulSet to manage log data.

1. Create the Elasticsearch Service:  
   Save the following manifest as es-svc.yaml:

text

apiVersion: v1  
kind: Service  
metadata:  
 name: elasticsearch  
 namespace: kube-logging  
spec:  
 selector:  
 app: elasticsearch  
 ports:  
 - port: 9200  
 name: rest  
 - port: 9300  
 name: inter-node  
 clusterIP: None # Headless service for StatefulSet

Apply the service:

bash

kubectl apply -f es-svc.yaml

1. Create the Elasticsearch StatefulSet:  
   Save the following manifest as es-sts.yaml:

text

apiVersion: apps/v1  
kind: StatefulSet  
metadata:  
 name: es-cluster  
 namespace: kube-logging  
spec:  
 serviceName: elasticsearch  
 replicas: 3  
 selector:  
 matchLabels:  
 app: elasticsearch  
 template:  
 metadata:  
 labels:  
 app: elasticsearch  
 spec:  
 containers:  
 - name: elasticsearch  
 image: docker.elastic.co/elasticsearch/elasticsearch:7.5.0  
 ports:  
 - containerPort: 9200  
 - containerPort: 9300  
 env:  
 - name: cluster.name  
 value: k8s-logs  
 volumeMounts:  
 - name: data  
 mountPath: /usr/share/elasticsearch/data  
 volumeClaimTemplates:  
 - metadata:  
 name: data  
 spec:  
 accessModes: ["ReadWriteOnce"]  
 resources:  
 requests:  
 storage: 3Gi # Adjust based on your needs

Apply the StatefulSet:

bash

kubectl apply -f es-sts.yaml

## Step 4: Deploy Kibana

Kibana will be deployed to visualize logs stored in Elasticsearch.

1. Create the Kibana Deployment:  
   Save the following manifest as kibana-deployment.yaml:

text

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: kibana  
 namespace: kube-logging  
spec:  
 replicas: 1  
 selector:  
 matchLabels:  
 app: kibana  
 template:  
 metadata:  
 labels:  
 app: kibana  
 spec:  
 containers:  
 - name: kibana  
 image: docker.elastic.co/kibana/kibana:7.5.0  
 ports:  
 - containerPort: 5601  
 env:  
 - name: ELASTICSEARCH\_URL  
 value: <http://elasticsearch.kube-logging.svc.cluster.local:9200> # Adjust based on your service URL structure.

Apply the deployment:

bash

kubectl apply -f kibana-deployment.yaml

## Step 5: Deploy Fluentd

Fluentd will collect logs from all nodes and forward them to Elasticsearch.

1. Create the Fluentd ConfigMap (for configuration):Save as fluentd-configmap.yaml:

text

apiVersion: v1  
kind: ConfigMap  
metadata:  
 name: fluentd-configmap  
 namespace: kube-logging   
data:  
 fluent.conf: |  
 @type forward   
 @log\_level info   
 <match kubernetes.\*\*>  
 @type elasticsearch   
 host elasticsearch   
 port 9200   
 logstash\_format true   
 index\_name fluentd   
 type\_name \_doc   
 </match>

1. Create Fluentd DaemonSet:

Save as fluentd-daemonset.yaml:

text

apiVersion: apps/v1   
kind: DaemonSet   
metadata:  
 name: fluentd   
 namespace: kube-logging   
spec:  
 selector:  
 matchLabels:  
 app: fluentd   
 template:  
 metadata:  
 labels:  
 app: fluentd   
 spec:  
 containers:  
 - name: fluentd   
 image: fluent/fluentd-kubernetes-daemonset:v1.8.0-debian-ubuntu-20.04   
 env:  
 - name: FLUENT\_ELASTICSEARCH\_HOST   
 value: "elasticsearch"  
 - name: FLUENT\_ELASTICSEARCH\_PORT   
 value : "9200"  
 volumeMounts:  
 - name : config-volume   
 mountPath : /fluentd/etc/fluent.conf   
 subPath : fluent.conf   
 volumes :  
 - name : config-volume   
 configMap :  
 name : fluentd-configmap

Apply the DaemonSet:

bash

kubectl apply -f fluentd-daemonset.yaml

## Step 6: Verify the Deployment

Check if all pods are running properly:

bash

kubectl get pods -n kube-logging

## Step 7: Access Kibana

To access Kibana, use port forwarding:

bash

kubectl port-forward deployment/kibana 5601 --namespace kube-logging

Now, open your browser and go to <http://localhost:5601>. You should see the Kibana interface.

## Conclusion

You have successfully set up an EFK stack on Kubernetes! You can now start collecting logs from your applications and visualize them using Kibana for better insights into your system's performance and health